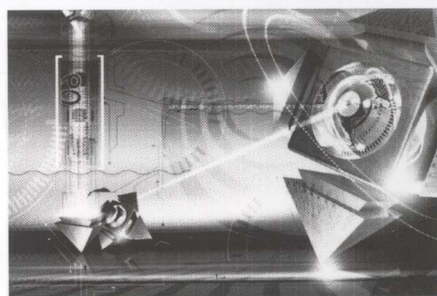


CLOUD COMPUTING & BIG DATA

TECHNICAL AND VOCATIONAL EDUCATION

工业和信息化人才培养规划教材

云计算与大数据技术



Cloud Computing & Big Data

王鹏 黄焱 安俊秀 张逸琴◎ 编著

系统讲授云计算和大数据技术

具体介绍集群计算技术和虚拟化技术

新技术、新应用的入门教材



人民邮电出版社
POSTS & TELECOM PRESS



精品系列

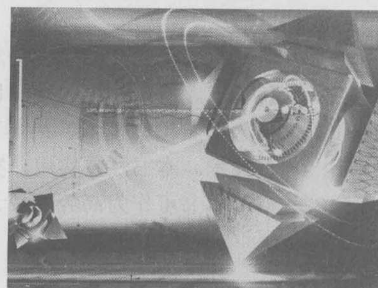


CLOUD COMPUTING & BIG DATA

TECHNICAL AND VOCATIONAL EDUCATION

工业和信息化人才培养规划教材

云计算与大数据技术



Cloud Computing & Big Data

王鹏 黄焱 安俊秀 张逸琴◎ 编著

人民邮电出版社
北京

图书在版编目（CIP）数据

云计算与大数据技术 / 王鹏等编著. -- 北京 : 人民邮电出版社, 2014.5
工业和信息化人才培养规划教材
ISBN 978-7-115-34803-6

Ⅰ. ①云… Ⅱ. ①王… Ⅲ. ①计算机网络—数据处理—教材 Ⅳ. ①TP393

中国版本图书馆CIP数据核字(2014)第032242号

内 容 提 要

本书全面介绍了云计算与大数据的基础知识、主要技术、基于集群技术的资源整合型云计算技术和基于虚拟化技术的资源切分型云计算技术。全书共 10 章，主要内容包括云计算与大数据概述、相关技术、虚拟化技术、集群系统基础、MPI、Hadoop、HPCC、Storm、数据中心技术和云计算大数据仿真技术。本书注重实用，实验丰富，将实验内容融合在课程内容中，使理论紧密联系实际。

本书可作为高等院校云计算、大数据相关课程的教材，也可作为相关技术人员的参考用书。

◆ 编　　著　王　鹏　黄　焱　安俊秀　张逸琴
　责任编辑　王　威
　责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　三河市潮河印业有限公司印刷

◆ 开本：787×1092　1/16
　印张：11.75　　　　2014 年 5 月第 1 版
　字数：303 千字　　　2014 年 5 月河北第 1 次印刷

定价：32.00 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316
反盗版热线：(010)81055315

前 言

计算技术的发展经历从合到分，又从分到合的历程，这一发展历程中内在的推动力就是技术。最早的电子管计算机系统价格昂贵、体积巨大，计算资源只能被集中放在机房，随着芯片技术的发展大规模集成电路技术使计算机的体积变得很小，同时微软视窗系统的出现使计算以前所未有的速度得到了普及，计算实现了由合到分的变化。分散的计算资源虽然给大家带来了方便但同时也带来了资源的浪费，而且在需要进行计算时又可能会出现资源不够的情况，这时网络技术的发展使计算资源再次被集中存放于机房成为了可能。

计算技术的发展特别是网络技术的发展催生了云计算技术的出现，云计算技术的出现被广泛地认为是信息技术的一次重大变革，大量的与云计算相关的软件和系统架构如雨后春笋般地出现。云计算技术将计算资源、存储资源以及相关各类广义的资源通过网络以服务的形式提供给资源的使用者，改变了传统信息技术架构中物理资源直接独占使用的模式，甚至从广义上讲只要是通过网络向用户提供服务的信息系统都被称为云计算系统。

云计算、物联网、社交网络的发展使人类社会的数据产生方式发生了变化，社会数据的规模正在以前所未有的速度增长，数据的种类五花八门，对海量、异构数据的存储、管理、分析和挖掘成为信息学科的热门领域，大数据技术逐渐进入人们的视野。

云计算与大数据出现以后随着进入这个领域的企业和研究机构的大量增加，对于云计算、大数据技术的认识出现了大量不同的定义。如果我们把云计算看作是一种通过网络实现资源服务的模式的话，则云计算技术可以被认为是实现云计算模式的所有技术的总称，这些技术包括虚拟化技术、分布式计算技术、分布式存储技术、网络技术等，不少技术是互联网时代就已存在的技术。大数据技术涵盖数据的存储、管理、分析和挖掘，这些技术并不是新的技术门类。

在云计算与大数据概念的内涵还没有完全得到业界的一致认识时，云计算与大数据产业的高速发展却十分出人预料，大量的客户需要企业提供相关的系统解决方案，一些地方希望能建设云计算中心，云计算与大数据人才的需求呈现出一种井喷的局面，不少学校都在规划建立云计算与大数据专业或开设相关课程以满足日益增长的人才需求。云计算与大数据课程专业该“学什么、如何学”正是本教材需要回答的问题。

信息技术这些年的高速发展使信息学科的整个格局也发生了变化。例如：在高性能计算领域已存在很久的集群技术在云计算和大数据时代再次成为系统架构的核心技术；在传统数据中心主机租赁业务中得到广泛应用的服务器虚拟技术，在云计算时代因为桌面虚拟化的大量使用得到了极大的发展，成为云计算技术的重要应用之一。

本书作为云计算与大数据技术的一本综合入门课程，我们一直在思考什么样的人才可以被称为云计算与大数据人才，培养的学生的知识结构是怎么样的，云计算与大数据作为一个高速发展的学科哪些知识是必须要了解的。从课程角度本书并不是对某一项技术的专门介绍，而是希望为学习云计算与大数据技术的同学提供一个完整的知识框架，为今后深入学习打下基础。

本书主要包含两大技术方向：集群计算技术和虚拟化技术，分别介绍了两个技术方向中学生需要了解的基础知识和典型系统，使学生在面对技术的快速度发展时能以不变应万变，避免出现在学校学的技术到工作岗位上时由于技术进步而用不上的问题。书中所介绍的相关知识和技术都有一定的普遍性和典型示范作用，在学习时重要的是要学习其中的系统思想。特别是我们在集群云计算系统中加入了基于消息传递机制高性能计算内容，消息传递机制揭示了集群系

统中节点间协调工作和数据传输模式的本质，不少人在学习云计算与大数据技术时知其然而不知其所以然的原因就在于不了解集群工作的基本机制。基于消息传递机制高性能计算知识虽然可能在实际中用得不多，但却可以使我们了解集群的基本工作机制。

云计算与大数据技术涉及面很广，本书在编写过程中参考并引用了大量前辈学者的研究成果和论述，对此编者向这些学者表示敬意，没有这些学者的努力本书是不可能完成的。云计算与大数据技术是一门高速发展的技术领域，新技术、新方法、新架构层出不穷，本书也是在不断探索和研究的新学科，由于作者的经验和能力所限，本书的结构、内容肯定存在许多疏漏和错误，望读者指正。

任课老师可以登录人民邮电出版社教材服务与资源网（www.ptpedu.com.cn）下载本书的PPT课件、教学大纲及实验相关资源，读者也可以登录本书支持网站 http://www.qhoa.org 或发送邮件至 jssyhuang@163.com 获取相关支持。

编　者

2014 年 2 月

目 录 CONTENTS

第5章 MPI——面向计算的高性能集群技术 66

第6章 Hadoop——分布式大数据系统 91

第7章 HPCC——面向数据的高性能计算集群系统 112

PART 1

第1章 云计算与大数据基础

1.1 云计算技术概述

1.1.1 云计算简介

云计算技术是硬件技术和网络技术发展到一定阶段而出现的一种新的技术模型，通常技术人员在绘制系统结构图时用一朵云的符号来表示网络，云计算这个奇怪的名字就是因此而得名的。云计算并不是对某一项独立技术的称呼，而是对实现云计算模式所需要的所有技术的总称。云计算技术的内容很多，包括分布式计算技术、虚拟化技术、网络技术、服务器技术、数据中心技术、云计算平台技术、存储技术等。从广义上说，云计算技术几乎包括了当前信息技术中的绝大部分。

维基百科中对云计算的定义为：云计算是一种基于互联网的计算方式，通过这种方式，共享的软硬件资源和信息可以按需求提供给计算机和其他设备。

2012年的国务院政府工作报告将云计算作为国家战略性新兴产业给出了定义：云计算是基于互联网的服务的增加、使用和交付模式，通常涉及通过互联网来提供动态、易扩展且经常是虚拟化的资源。云计算是传统计算机和网络技术发展融合的产物，它意味着计算能力也可作为一种商品通过互联网进行流通。

对于以上的定义我们可以将云计算从非技术的角度理解为一种通过网络的资源整合输出模式，只要是为了达到资源整合输出这个目的的技术都可以被称为云计算技术。从定义中也可以看出网络在云计算技术中的重要性，如果没有网络的高速发展，则云计算这种模式是无法实现的。

云计算技术的出现改变了信息产业传统的格局。传统的信息产业企业既是资源的整合者又是资源的使用者，这就像一个电视机企业既要生产电视机还要生产发电机一样，这种格局并不符合现代产业分工高度专业化的需求，同时也不符合企业需要灵敏地适应客户的需要。传统的计算资源和存储资源大小通常是相对固定的，面对客户高波动性的需求时会非常不敏捷，企业的计算和存储资源要么是被浪费，要么是面对客户峰值需求时力不从心。云计算技术使资源与用户需求之间是一种弹性化的关系，资源的整合者和资源的使用者并不是一个企业，资源的使

用者只需要对资源按需付费，从而敏捷地响应客户不断变化的资源需求，这一方法降低了资源使用者的成本，提高了资源的利用效率。

在云计算时代基本的3种角色为：资源的整合运营者、资源的使用者、终端客户。资源的整合运营者就像是发电厂负责资源的整合输出，资源的使用者负责将资源转变为满足客户需求的各种应用，终端客户为资源的最终消费者。

云计算这种新的模式的出现被认为是信息产业的一大变革，从而吸引了大量企业的注意力。国际巨头IBM、微软、谷歌、DELL等企业都在云计算领域进行了全面的布局，变革之时正是机会出现的时候，云计算的出现更是给国内企业一次重新布局的机会，可以看到国内的华为、中兴、腾讯、阿里、联想、浪潮、五舟等企业都相继提出自己的云计算战略规划，并在云计算技术和市场都进行了全面的布局。

云计算技术作为一项涵盖面广且对产业影响深远的技术，未来将逐步渗透到信息产业和其他产业的方方面面，并将深刻改变产业的结构模式、技术模式和产品销售模式，进而深刻影响人们的生活。云计算会逐步成为人们生活中必不可少的技术。同时移动互联网的出现使云计算应用走向了人们的指间，推动了云计算技术的应用发展，今后云计算将是一项随时、随地、随身为我们提供服务的技术。云计算的出现也将如电的出现一般，为信息产业的发展提供无限的想象空间，使应用的创新能力得到完全释放。

1.1.2 云计算的特点

为了理解云计算这个概念，只了解一个简单的定义是不够的，我们还需要利用云计算技术的特点来判断一个技术是否是云计算技术。与传统的资源提供方向相比，云计算具有以下特点。

（1）资源池弹性可扩张。

云计算系统的一个重要特征就是资源的集中管理和输出，这就是所谓的资源池。从资源低效率的分散使用到资源高效的集约化使用正是云计算的基本特征之一。分散的资源使用方法造成了资源的极大浪费，现在每个人都可能有一到两台自己的计算机，但对这种资源的利用率却非常的低，计算机在大量时间都是在等待状态或是在处理文字数据等低负荷的任务。资源集中起来后资源的利用效率会大大地提高，随着资源需求的不断提高，资源池的弹性化扩张能力成为云计算系统的一个基本要求，云计算系统只有具备了资源的弹性化扩张能力才能有效地应对不断增长的资源需求。大多数云计算系统都能较为方便地实现新资源的加入。

（2）按需提供资源服务。

云计算系统带给客户最重要的好处就是敏捷地适应用户对资源不断变化的需求，云计算系统实现按需向用户提供资源能大大节省用户的硬件资源开支，用户不用自己购买并维护大量固定的硬件资源，只需向自己实际消费的资源量来付费。按需提供资源服务使应用开发者在逻辑上可以认为资源池的大小是不受限制的，这就使应用软件的开发者拥有了更大的想象空间和创新空间，更多的有趣应用将在云计算时代被创造出来，应用开发者的主要精力只需要集中在自己的应用上。

（3）虚拟化。

现有的云计算平台的重要特点是利用软件来实现硬件资源的虚拟化管理、调度及应用。通过虚拟平台用户使用网络资源、计算资源、数据库资源、硬件资源、存储资源等，与在自己的本地计算机上使用的感觉是一样的，相当于是在操作自己的计算机，而在云计算中利用虚拟化技术可大大降低维护成本和提高资源的利用率。

（4）网络化的资源接入。

从最终用户的角度看，基于云计算系统的应用服务通常都是通过网络来提供的，应用开发者将云计算中心的计算、存储等资源封装为不同的应用后往往会通过网络提供给最终的用户。云计算技术必须实现资源的网络化接入才能有效地向应用开发者和最终用户提供资源服务。这就像有了发电厂必须还要有输电线才能将电传送给用户。所以网络技术的发展是推动云计算技术出现的首要动力。目前一些企业将网络化的软件和硬件都称为云计算，就是因为网络化的资源接入方式是从最终用户角度能看到的云计算的重要特征之一，这些产品的称呼不一定准确但却是对云计算特征的反映。

（5）高可靠性和安全性。

用户数据存储在服务器端，而应用程序在服务器端运行，计算由服务器端来处理。所有的服务分布在不同的服务器上，如果什么地方（节点）出问题就在什么地方终止它，另外再启动一个程序或节点，即自动处理失败节点，从而保证了应用和计算的正常进行。

数据被复制到多个服务器节点上有多个副本（备份），存储在云里的数据即使遇到意外删除或硬件崩溃也不会受到影响。

1.1.3 云计算技术分类

目前已出现的云计算技术种类非常多，对于云计算的分类可以有多种角度。从技术路线角度可以分为资源整合型云计算和资源切分型云计算；从服务对像角度可以分为公有云和私有云；按资源封装的层次可以分为基础设施即服务（Infrastructure as a Service，IaaS）、平台即服务（Platform as a Service，PaaS）和软件即服务（Software as a Service，SaaS）。

1．按技术路线分类

资源整合型云计算：这种类型的云计算系统在技术实现方面大多体现为集群架构，通过将大量节点的计算资源和存储资源整合后输出。这类系统通常能实现跨节点弹性化的资源池构建，核心技术为分布式计算和存储技术。MPI、Hadoop、HPCC、Storm 等都可以被分类为资源整合型云计算系统。

资源切分型云计算：这种类型最为典型的就是虚拟化系统，这类云计算系统通过系统虚拟化实现对单个服务器资源的弹性化切分，从而有效地利用服务器资源，其核心技术为虚拟化技术，这种技术的优点是用户的系统可以不做任何改变接入采用虚拟化技术的云系统，是目前应用较为广泛的技术，特别是在桌面云计算技术上应用得较为成功，缺点是跨节点的资源整合代价较大。KVM、VMware 都是这类技术的代表。

2．按服务对象分类

公有云：指服务对象是面向公众的云计算服务，公有云对云计算系统的稳定性、安全性和并发服务能力有更高的要求。

私有云：指主要服务于某一组织内部的云计算服务，其服务并不向公众开放，如企业、政府内部的云服务。

公有云与私有云的界限并不是特别清晰，有时服务于一个地区和团体的云也被称为公有云。所以这种云计算分类方法并不是一种准确的分类方法，主要是在商业领域的一种称呼。

3．按资源封装的层次分类

基础设施即服务（Infrastructure as a Service，IaaS）：把单纯的计算和存储资源不经封装地直接通过网络以服务的形式提供的用户使用。这类云计算服务用户的自主性较大，就像是发电厂将发的电直接送出去一样。这类云服务的对象往往是具有专业知识能力的资源使用者，传统数据中心的主机租用等可能作为 IaaS 的典型代表。

平台即服务（Platform as a Service，PaaS）：计算和存储资源经封装后，以某种接口和协议的形式提供给用户调用，资源的使用者不再直接面对底层资源。平台即服务需要平台软件的支撑，可以认为是从资源到应用软件的一个中间件，通过这类中间件可以大大减小应用软件开发时的技术难度。这类云服务的对象往往是云计算应用软件的开发者，平台软件的开发需要使用者具有一定的技术能力。

软件即服务（Software as a Service，SaaS）：将计算和存储资源封装为用户可以直接使用的应用并通过网络提供给用户，SaaS 面向的服务对象为最终用户，用户只是对软件功能进行使用，无需了解任何云计算系统的内部结构，也不需要用户具有专业的技术开发能力。

如图 1.1 所示，云计算系统按资源封装的层次分为 IaaS、PaaS、SaaS，分别为对底层硬件资源不同级别的封装，从而实现将资源转变为服务的目的。传统的信息系统资源的使用者通常是以直接占有物理硬件资源的形式来使用资源的，而云计算系统通过 IaaS、PaaS、SaaS 等不同层次的封装将物理硬件资源封装后，以服务的形式利用网络提供给资源的使用者。在这里资源的使用者可能是资源的二次加工者，也可能是最终应用软件的使用者，通常 IaaS、PaaS 层面向的资源使用者往往是资源的二次加工者，这类资源的使用者并不是资源的最终消费者，他们将资源转变为应用服务程序后以 SaaS 的形式提供给资源的最终消费者。实现对物理资源封装的技术并不是惟一的，目前不少的软件都能实现，甚至有的系统只有 SaaS 层，并没有进行逐层的封装。

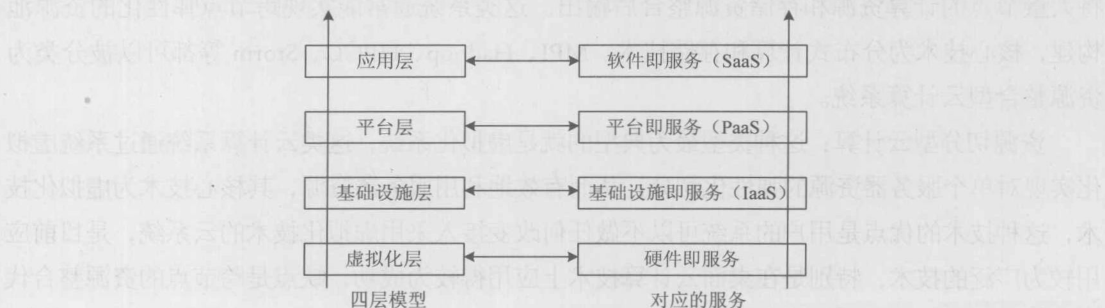


图 1.1　云计算服务体系结构

云计算的服务层次是根据服务类型即服务集合来划分，与大家熟悉的计算机网络体系结构中层次的划分不同。在计算机网络中每个层次都实现一定的功能，层与层之间有一定关联。而云计算体系结构中的层次是可以分割的，即某一层次可以单独完成一项用户的请求而不需要其他层次为其提供必要的服务和支持。

在云计算服务体系结构中各层次与相关云产品对应。

应用层对应 SaaS 软件即服务如：Google APPS、SoftWare+Services。

平台层对应 PaaS 平台即服务如：IBM IT Factory、Google APPEngine、Force.com。

基础设施层对应 IaaS 基础设施即服务如：Amazo EC2、IBM Blue Cloud、Sun Grid。

虚拟化层对应硬件即服务结合 PaaS 提供硬件服务，包括服务器集群及硬件检测等服务。

1.2 大数据技术概述

1.2.1 大数据简介

计算和数据是信息产业不变的主题，在信息和网络技术迅速发展的推动下，人们的感知、计算、仿真、模拟、传播等活动产生了大量的数据，数据的产生不受时间、地点的限制，大数据的概念逐渐形成，大数据涵盖了计算和数据两大主题，是产业界和学术界的研究热点，被誉为未来十年的革命性技术。

2008 年，《Nature》杂志推出了“大数据”专辑，引发了学术界和产业界的关注；2011 年，大数据应用进入我国并快速发展，目前大数据的应用和研究已经是学术界和产业界的热点；2012 年 3 月，美国政府发布《大数据研究和发展倡议》，投资 2 亿美元发展大数据，用以强化国土安全、转变教育学习模式、加速科学和工程领域的创新速度和水平；2012 年 7 月，日本提出以电子政府、电子医疗、防灾等为中心制定新 ICT（信息通信技术）战略，发布“新 ICT 计划”，重点关注大数据研究和应用；2013 年 1 月，英国政府宣布将在对地观测、医疗卫生等大数据和节能计算技术方面投资 1.89 亿英镑；2013 年我国上海、重庆等地相继发布大数据行动计划。

1．什么是大数据

大数据是一个比较抽象的概念，维基百科将大数据描述为：大数据是现有数据库管理工具和传统数据处理应用很难处理的大型、复杂的数据集，大数据的挑战包括采集、存储、搜索、共享、传输、分析和可视化等。

大数据的“大”是一个动态的概念，以前 10GB 的数据是个天文数字；而现在，在地球、物理、基因、空间科学等领域，TB 级的数据集已经很普遍。大数据系统需要满足以下三个特性。

（1）规模性（Volume）：需要采集、处理、传输的数据容量大；

（2）多样性（Variety）：数据的种类多、复杂性高；

（3）高速性（Velocity）：数据需要频繁地采集、处理并输出。

2．数据的来源

大数据的数据来源很多，主要有信息管理系统、网络信息系统、物联网系统、科学实验系统等，其数据类型包括结构化数据、半结构化数据和非结构化数据。

（1）管理信息系统：企业内部使用的信息系统，包括办公自动化系统、业务管理系统等，是常见的数据产生方式。管理信息系统主要通过用户输入和系统的二次加工的方式生成数据，其产生的数据大多为结构化数据，存储在数据库中。

（2）网络信息系统：基于网络运行的信息系统是大数据产生的重要方式，电子商务系统、社交网络、社会媒体、搜索引擎等都是常见的网络信息系统，网络信息系统产生的大数据多为半结构化或无结构化的数据，网络信息系统与管理信息系统的区别在于管理信息系统是内部使用的，不接入外部的公共网络。

（3）物联网系统：通过传感器获取外界的物理、化学、生物等数据信息。

（4）科学实验系统：主要用于学术科学研究，其环境是预先设定的，数据既可以是由真实实验产生的也可以是通过模拟方式获取仿真的。

3．生产数据的三个阶段

数据库技术诞生以来，人们生产数据的方式经过了三个主要的发展阶段。

（1）被动式生成数据：数据库技术使得数据的保存和管理变得简单，业务系统在运行时产生的数据直接保存数据库中，这个时候数据的产生是被动的，数据是随着业务系统的运行产生的。

（2）主动式生成数据：互联网的诞生尤其是 Web 2.0、移动互联网的发展大大加速了数据的产生，人们可以随时随地通过手机等移动终端随时随地地生成数据，人们开始主动地生成数据。据统计，在 1min 的时间内，新浪平均有 2 万条微博产生，苹果商店平均有 4.7 万次应用下载，淘宝平均有 6 万件商品交易记录，百度大约产生了 90 万次的搜索查询，数据的生成大大加速。

（3）感知式生成数据：感知技术尤其是物联网的发展促进了数据生成方式发生了根本性的变化，遍布在城市各个角落的摄像头等数据采集设备源源不断地自动采集、生成数据。

4．大数据的特点

在大数据的背景下，数据的采集、分析、处理与传统方式有很大的不同。

（1）数据产生方式：在大数据时代，数据的产生方式发生了巨大的变化，数据的采集方式由以往的被动采集数据转变为主动生成数据。

（2）数据采集密度：以往我们进行数据采集时的采样密度较低，获得的采样数据有限；在大数据时代，有了大数据处理平台的支撑，我们可以对需要分析的事件的数据进行更加密集地采样，从而精确地获取事件的全局数据。

（3）数据源：以往我们多从各个单一的数据源获取数据，获取的数据较为孤立，不同数据源之间的数据整合难度较大；在大数据时代，我们可以通过分布式计算、分布式文件系统、分布式数据库等技术对多个数据源获取的数据进行整合处理。

（4）数据处理方式：以往我们对数据的处理大多采用离线处理的方式，对已经生成的数据集中进行分析处理，不对实时产生的数据进行分析；在大数据时代，我们可以根据应用的实际需求对数据采取灵活的处理方式，对于较大的数据源、响应时间要求低的应用可以采取批处理的方式进行集中计算，而对于响应时间要求高的实时数据处理则采用流处理的方式进行实时计算，并且可以通过对历史数据的分析进行预测分析。

大数据需要处理的数据大小通常达到 PB（1024 TB）或 EB（1024 PB）级，数据的类型多种多样，包括结构化数据、半结构化数据和非结构化数据。巨大的数据量和种类繁多的数据类型给大数据系统的存储和计算带来很大挑战，单节点的存储容量和计算能力成为瓶颈。

分布式系统是对大数据进行处理的基本方法，分布式系统将数据切分后存储到多个节点上，并在多个节点上发起计算，解决单节点的存储和计算瓶颈。常见的数据切分的方法有随机方法、哈希方法和区间方法，随机方法将数据随机分布到不同的节点，哈希方法根据数据的某一行或者某一列的哈希值将数据分布到不同的节点，区间方法将不同的数据按照不同区间分布到不同节点。

5．大数据的应用领域

大数据在社会生活的各个领域得到广泛的应用，如科学计算、金融、社交网络、移动数据、物联网、网页数据、多媒体等，不同领域的大数据应用具有不同的特点，其对响应时间、系统稳定性、计算精确性的要求各不相同，其对比如表 1.1 所示。

表 1.1　典型的大数据应用特征对比

应用领域	示例	用户数量	响应时延	数据量级	稳定性	精确度
科学计算	基因计算	小	长	TB	一般	非常高
金融	股票交易	大	实时	GB	非常高	非常高
社交网络	Facebook	非常大	快速	PB	高	高
移动数据	移动终端	非常大	快速	TB	高	高
物联网	传感网	大	快速	TB	高	高
网页数据	新闻网站	非常大	快速	GB	高	高
多媒体	视频网站	非常大	快速	GB	高	一般

1.2.2 主要的大数据处理系统

大数据处理的数据源类型多种多样，如结构化数据、半结构化数据、非结构化数据，数据处理的需求各不相同，有些场合需要对海量已有数据进行批量处理，有些场合需要对大量的实时生成的数据进行实时处理，有些场合需要在进行数据分析时进行反复迭代计算，有些场合需要对图数据进行分析计算。目前主要的大数据处理系统有数据查询分析计算系统、批处理系统、流式计算系统、迭代计算系统、图计算系统和内存计算系统。

1. 数据查询分析计算系统

大数据时代，数据查询分析计算系统需要具备对大规模数据进行实时或准实时查询的能力，数据规模的增长已经超出了传统关系型数据库的承载和处理能力。目前主要的数据查询分析计算系统包括 HBase、Hive、Cassandra、Dremel、Shark、Hana 等。

HBase：开源、分布式、面向列的非关系型数据库模型，是 Apache 的 Hadoop 项目的子项目，源于 Google 论文《Bigtable：一个结构化数据的分布式存储系统》，实现了其中的压缩算法、内存操作和布隆过滤器。HBase 的编程语言为 Java。HBase 的表能够作为 MapReduce 任务的输入和输出，可以通过 Java API 来存取数据。

Hive：基于 Hadoop 的数据仓库工具，用于查询、管理分布式存储中的大数据集，提供完整的 SQL 查询功能，可以将结构化的数据文件映射为一张数据表。Hive 提供了一种类 SQL 语言（HiveQL）可以将 SQL 语句转换为 MapReduce 任务运行。

Cassandra：开源 NoSQL 数据库系统，最早由 Facebook 开发，并于 2008 年开源，由于其良好的可扩展性，Cassandra 被 Facebook、Twitter、Rackspace、Cisco 等公司使用，其数据模型借鉴了 Amazon 的 Dynamo 和 Google BigTable，是一种流行的分布式结构化数据存储方案。

Impala：由 Cloudera 公司主导开发，是运行在 Hadoop 平台上的开源的大规模并行 SQL 查询引擎。用户可以使用标准的 SQL 接口的工具查询存储在 Hadoop 的 HDFS 和 HBase 中的 PB 级大数据。

Shark：Spark 上的数据仓库实现，即 SQL on Spark，与 Hive 相兼容，但处理 Hive QL 的性能比 Hive 快 100 倍。

Hana：由 SAP 公司开发的与数据源无关、软硬件结合、基于内存计算的平台。

2. 批处理系统

MapReduce 是被广泛使用的批处理计算模式。MapReduce 对具有简单数据关系、易于划分的大数据采用“分而治之”的并行处理思想，将数据记录的处理分为 Map 和 Reduce 两个简单的抽象操作，提供了一个统一的并行计算框架。批处理系统将复杂的并行计算的实现进行封装，大大降低开发人员的并行程序设计难度。Hadoop 和 Spark 是典型的批处理系统。MapReduce 的批处理模式不支持迭代计算。

Hadoop：目前大数据处理最主流的平台，是 Apache 基金会的开源软件项目，使用 Java 语言开发实现。Hadoop 平台使开发人员无需了解底层的分布式细节，即可开发出分布式程序，在

集群中对大数据进行存储、分析。

Spark：由加州伯克利大学 AMP 实验室开发，适合用于机器学习、数据挖掘等迭代运算较多的计算任务。Spark 引入了内存计算的概念，运行 Spark 时服务器可以将中间数据存储在 RAM 内存中，大大加速数据分析结果的返回速度，可用于需要互动分析的场景。

3．流式计算系统

流式计算具有很强的实时性，需要对应用源源不断产生的数据实时进行处理，使数据不积压、不丢失，常用于处理电信、电力等行业应用以及互联网行业的访问日志等。Facebook 的 Scribe、Apache 的 Flume、Twitter 的 Storm、Yahoo 的 S4、UCBerkeley 的 Spark Streaming 是常用的流式计算系统。

Scribe：Scribe 由 Facebook 开发开源系统，用于从海量服务器实时收集日志信息，对日志信息进行实时的统计分析处理，应用在 Facebook 内部。

Flume：Flume 由 Cloudera 公司开发，其功能与 Scribe 相似，主要用于实时收集在海量节点上产生的日志信息，存储到类似于 HDFS 的网络文件系统中，并根据用户的需求进行相应的数据分析。

Storm：基于拓扑的分布式流数据实时计算系统，由 BackType 公司（后被 Twitter 收购）开发，现已经开放源代码，并应用于淘宝、百度、支付宝、Groupon、Facebook 等平台，是主要的流数据计算平台之一。

S4：S4 的全称是 Simple Scalable Streaming System，是由 Yahoo 开发的通用、分布式、可扩展、部分容错、具备可插拔功能的平台，其设计目的是根据用户的搜索内容计算得到相应的推荐广告，现已经开源，是重要的大数据计算平台。

Spark Streaming：构建在 Spark 上的流数据处理框架，将流式计算分解成一系列短小的批处理任务进行处理。网站流量统计是 Spark Streaming 的一种典型的使用场景，这种应用既需要具有实时性，还需要进行聚合、去重、连接等统计计算操作，如果使用 Hadoop MapReduce 框架，则可以很容易地实现统计需求，但无法保证实时性；如果使用 Storm 这种流式框架则可以保证实时性，但实现难度较大；Spark Streaming 可以以准实时的方式方便地实现复杂的统计需求。

4．迭代计算系统

针对 MapReduce 不支持迭代计算的缺陷，人们对 Hadoop 的 MapReduce 进行了大量改进，Haloop、iMapReduce、Twister、Spark 是典型的迭代计算系统。

HaLoop：Haloop 是 Hadoop MapReduce 框架的修改版本，用于支持迭代、递归类型的数据分析任务，如 PageRank、K-means 等。

iMapReduce：一种基于 MapReduce 的迭代模型，实现了 MapReduce 的异步迭代。

Twister：基于 Java 的迭代 MapReduce 模型，上一轮 Reduce 的结果会直接传送到下一轮的 Map。

Spark：基于内存计算的开源集群计算框架。

5．图计算系统

社交网络、网页链接等包含具有复杂关系的图数据，这些图数据的规模巨大，可包含数十亿顶点和上百亿条边，图数据需要由专门的系统进行存储和计算。常用的图计算系统有 Google

公司的 Pregel、Pregel 的开源版本 Giraph、微软的 Trinity、Berkeley AMPLab 的 GraphX 以及高速图数据处理系统 PowerGraph。

Pregel：Google 公司开发的一种面向图数据计算的分布式编程框架，采用迭代的计算模型。Google 的数据计算任务中，大约 80%的任务处理采用 MapReduce 模式，如网页内容索引；图数据的计算任务约占 20%，采用 Pregel 进行处理。

Giraph：一个迭代的图计算系统，最早由雅虎公司借鉴 Pregel 系统开发，后捐赠给 Apache 软件基金会，成为开源的图计算系统。Giraph 是基于 Hadoop 建立的，Facebook 在其脸谱搜索服务中大量使用了 Giraph。

Trinity：微软公司开发的图数据库系统，该系统是基于内存的数据存储与运算系统，源代码不公开。

GraphX：由 AMPLab 开发的运行在数据并行的 Spark 平台上的图数据计算系统。

PowerGraph：高速图处理系统，常用于广告推荐计算和自然语言处理。

6．内存计算系统

随着内存价格的不断下降、服务器可配置内存容量的不断增长，使用内存计算完成高速的大数据处理已成为大数据处理的重要发展方向。目前常用的内存计算系统有分布式内存计算系统 Spark、全内存式分布式数据库系统 HANA、Google 的可扩展交互式查询系统 Dremel。

Dremel：Google 的交互式数据分析系统，可以在数以千计的服务器组成的集群上发起计算，处理 PB 级的数据。Dremel 是 Google MapReduce 的补充，大大缩短了数据的处理时间，成功地应用在 Google 的 bigquery 中。

HANA：SAP 公司开发的基于内存技术、面向企业分析性的产品。

Spark：基于内存计算的开源集群计算系统。

1.2.3 大数据处理的基本流程

大数据的处理流程可以定义为在适合工具的辅助下，对广泛异构的数据源进行抽取和集成，结果按照一定的标准统一存储，利用合适的数据分析技术对存储的数据进行分析，从中提取有益的知识并利用恰当的方式将结果展示给终端用户。大数据处理的基本流程如图 1.2 所示。

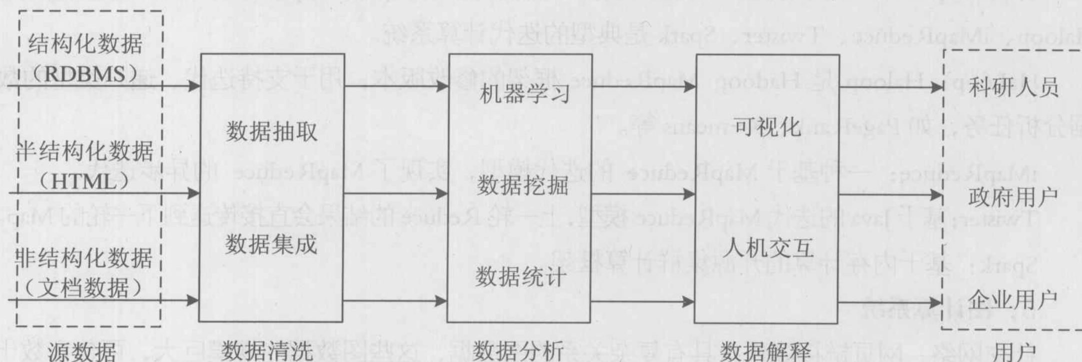


图 1.2 大数据处理的基本流程

1．数据抽取与集成

由于大数据处理的数据来源类型丰富，大数据处理的第一步是对数据进行抽取和集成，从中提取出关系和实体，经过关联和聚合等操作，按照统一定义的格式对数据进行存储。现有的数据抽取和集成方法有3种：基于物化或ETL方法的引擎（Materialization or ETL Engine）、基于联邦数据库或中间件方法的引擎（Federation Engine or Mediator）、基于数据流方法的引擎（Stream Engine）。

2．数据分析

数据分析是大数据处理流程的核心步骤，通过数据抽取和集成环节，我们已经从异构的数据源中获得了用于大数据处理的原始数据，用户可以根据自己的需求对这些数据进行分析处理，如数据挖掘、机器学习、数据统计等，数据分析可以用于决策支持、商业智能、推荐系统、预测系统等。

3．数据解释

大数据处理流程中用户最关心的是数据处理的结果，正确的数据处理结果只有通过合适的展示方式才能被终端用户正确理解，因此数据处理结果的展示非常重要，可视化和人机交互是数据解释的主要技术。

我们在开发调试程序的时候经常通过打印语句的方式来呈现结果，这种方式非常灵活、方便，但只有熟悉程序的人才能很好地理解打印结果。

使用可视化技术，可以将处理的结果通过图形的方式直观地呈现给用户，标签云（T ag Cloud）、历史流（History Flow）、空间信息流（Spatial Information Flow）等是常用的可视化技术，用户可以根据自己的需求灵活地使用这些可视化技术；人机交互技术可以引导用户对数据进行逐步的分析，使用户参与到数据分析的过程中，使用户可以深刻地理解数据分析结果。

1.3 云计算与大数据的发展

1．云计算与大数据发展历程

很多人认为云计算是在近些年才被提出来的，其实早在1958年，人工智能之父John McCarthy发明了函数式语言LISP，LISP语言后来成为MapReduce的思想来源。1960年John McCarthy预言“今后计算机将会作为公共设施提供给公众”，这一概念与我们现在所定义的云计算已非常相似，但当时的技术条件决定了这一设想只是一种对未来技术发展的预言。云计算在技术发展到一定阶段后才能真正出现。一般认为云计算是网络技术发展到一定阶段后必然出现的新的技术体系和产业模式。很难想象在1986年中国第一封E-mail发出去时560bps的网速条件下能出现云计算这样的技术变革。1984年SUN公司提出“网络就是计算机”这一具有云计算特征的论点，2006年Google公司CEO Eric Schmidt提出云计算概念，2008年云计算概念全面进入中国，2009年中国首届云计算大会召开，此后云计算技术和产品迅速地发展起来。

随着社交网络、物联网等技术的发展，数据正在以前所未有的速度增长和积累，IDC 的研究数据表明，全球的数据量每年增长 50%，两年翻一番，这意味着全球近两年产生的数据量将超过之前全部数据的总和。2011 年全球数据总量已达 1.8ZB，到 2020 年，全球数据总量将达到 35 ZB。2008 年《Nature》杂志推出了大数据专刊，2011 年《Science》杂志推出大数据专刊，讨论科学研究的中大数据问题。2012 年大数据的关注度和影响力快速增长，成为当年达沃斯世界经济论坛的主题，美国政府启动大数据发展计划。中国计算机学会于 2012 年成立了大数据专家委员会，并发布了大数据技术白皮书。

图 1.3 所示为云计算、大数据两个关键词近年来的网络关注度，可以看出 2012 年至今大数据的关注度越来越高，云计算和大数据是信息技术未来的发展方向。

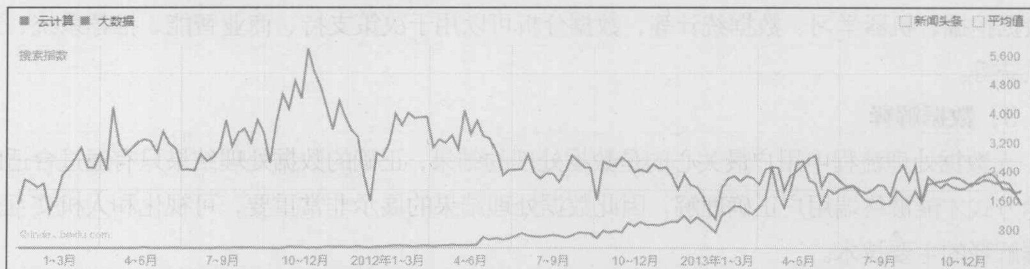


图 1.3　近年来 云计算、大数据的关注度

网络技术在云计算和大数据的发展历程中发挥了重要的推动作用。可以认为信息技术的发展经历了硬件发展推动和网络技术推动两个阶段。早期主要以硬件发展为主要动力，在这个阶段硬件的技术水平决定着整个信息技术的发展水平，硬件的每一次进步都有力地推动着信息技术的发展，从电子管技术到晶体管技术再到大规模集成电路，这种技术变革成为产业发展的核心动力。但网络技术的出现逐步地打破了单纯的硬件能力决定技术发展的格局，通信带宽的发展为信息技术的发展提供了新的动力，在这一阶段通信带宽成为了信息技术发展的决定性力量之一，云计算、大数据技术的出现正是这一阶段的产物，其广泛应用并不是单纯靠某一个人发明而是由于技术发展到现在的必然产物，生产力决定生产关系的规律在这里依然是成立的。

当前移动互联网的出现并迅速普及更是对云计算、大数据的发展起到了推动作用。移动瘦客户终端与云计算资源池的结合大大拓展了移动应用的思路，云计算资源得以在移动终端上实现随时、随地、随身资源服务。移动互联网再次拓展了以网络化资源交付为特点的云计算技术的应用能力，同时也改变了数据的产生方式，推动了全球数据的快速增长，推动了大数据的技术和应用的发展。

云计算是一种全新的领先信息技术，结合 IT 技术和互联网实现超级计算和存储的能力，而推动云计算兴起的动力是高速互联网和虚拟化技术的发展、更加廉价且功能强劲的芯片及硬盘、数据中心的发展。云计算作为下一代企业数据中心，其基本形式为大量链接在一起的共享 IT 基础设施，不受本地和远程计算机资源的限制，可以很方便地访问云中的“虚拟”资源，使用户和云服务提供商之间可以像访问网络一样进行交互操作。具体来讲，云计算的兴起有以下因素。

（1）高速互联网技术发展。

网络用于信息发布、信息交换、信息收集、信息处理。网络内容不再像早些年那样是静态的，门户网站随时在更新着网站中的内容，网络的功能、网络速度也在发生巨大的变化，网络成为人们学习、工作和生活的一部分。不过网站只是云计算应用和服务的缩影，云计算强大的功能正在移动互联网、大数据时代崭露头角。

云计算能够利用现有的 IT 基础设施在极短的时间内处理大量的信息以满足动态网络的高性能的需求。

（2）资源利用率需求。

能耗是企业特别关注的问题。大多数企业服务器的计算能力使用率很低，但同样需要消耗大量的能源进行数据中心降温。引入云计算模式后可以通过整合资源或采用租用存储空间、租用计算能力等服务来降低企业运行成本和节省能源。

同时，利用云计算将资源集中，统一提供可靠服务，并能减少企业成本，提升企业灵活性，企业可以把更多的时间用于服务客户和进一步研发新的产品上。

（3）简单与创新需求。

在实际的业务需求中，越来越多的个人用户和企业用户都在期待着使用计算机操作能简单化，能够直接通过购买软件或硬件服务而不是软件或硬件实体，为自己的学习、生活和工作带来更多的便利，能在学习场所、工作场所、住所之间建立便利的文件或资料共享的纽带。而对资源的利用可以简化到通过接入网络就可以实现自己想要实现的一切，就需要在技术上有所创新，利用云计算来提供这一切，将我们需要的资料、数据、文档、程序等全部放在云端实现同步。

（4）其他需求。

连接设备、实时数据流、SOA 的采用以及搜索、开放协作、社会网络和移动商务等的移动互联网应用急剧增长，数字元器件性能的提升也使 IT 环境的规模大幅度提高，从而进一步加强了对一个由统一的云进行管理的需求。

个人或企业希望按需计算或服务，能在不同的地方实时实现项目、文档的协作处理，能在繁杂的信息中方便地找到自己需要的信息等需求也是云计算兴起的原因之一。

人类历史不断地证明生产力决定生产关系，技术的发展历史也证明了技术能力决定技术的形态，纵观整个信息技术的发展历史（见图 1.4），从图中可以看出信息产业发展有两个重要的内在动力在不同时期起着作用：硬件驱动力、网络驱动力。这两种驱动力量的对比和变化决定着产业中不同产品的出现时期以及不同形态的企业出现和消亡的时间。也正是这两种驱动力的力量变化造成了信息产业技术体系的分分合合，技术的形态也经历了从合到分和从分到合的两个过程，由最早集中的计算到个人计算机分散的计算再到集中的云计算。整个信息产业中出现的各种产品模式和企业模式都能在图中找到位置，这幅图既能解释产业历史又能预测产业未来，是我们解开很多产业困惑的钥匙。

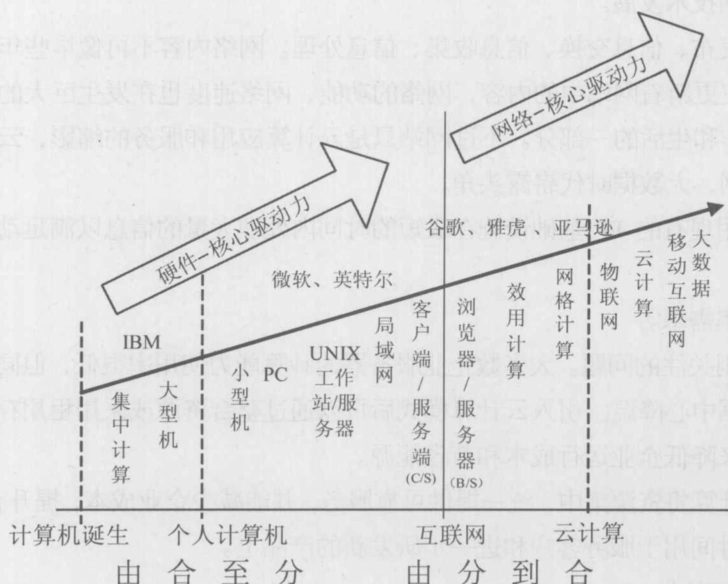


图 1.4 信息产业发展演进路线

硬件驱动的时代诞生了 IBM、微软、Intel 等企业。20 世纪 50 年代最早的网络开始出现，信息产业的发展驱动力中开始出现网络的力量，但当时网络性能很弱，网络并不是推动信息产业发展的主要动力，处理器等硬件的影响还占绝对主导因素。但随着网络的发展，网络通信带宽逐步加大，从 20 世纪 80 年代的局域网到 20 世纪 90 年代的互联网，网络逐渐成为了推动信息产业发展的主导力量，这个时期诞生了百度、谷歌、亚马逊等企业。直到云计算的出现才标志着网络已成为信息产业发展的主要驱动力，此时技术的变革即将出现。

2．为云计算与大数据发展做出贡献的科学家

在云计算与大数据的发展过程中不少科学家都做出了重要的贡献，让我们向这些科学家表示崇高的敬意。

- 超级计算机之父——西摩·克雷（Seymour Cray）（见图 1.5）

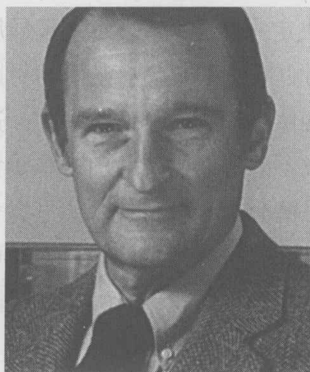


图 1.5 西摩·克雷

在人类解决计算和存储问题的历程中，西摩·克雷成为了一座丰碑，被称为超级计算机之父。西摩·克雷，生于1925年9月28日，美国人，1958年设计建造了世界上第一台基于晶体管的超级计算机，成为计算机发展史上的重要里程碑。同时也对精简指令（RISC）高端微处理器的产生 有重大的贡献。1972年，他创办了克雷研究公司，公司的宗旨是只生产超级计算机。此后的十余年中，克雷先后创造了 Cray-1、Cray-2 等机型。作为高性能计算机领域中最重要的人物之一，他亲手设计了 Cray 全部的硬件与操作系统。Cray 机成为了从事高性能计算学者中永远的记忆，截至1986年1月，世界上有130台超级计算机投入使用，其中大约90台是由克雷的上市公司——克雷研究所研制的。美国的《商业周刊》在1990年的一篇文章中曾这样写道："西摩·克雷的天赋和非凡的干劲已经给本世纪的技术留下了不可磨灭的印记"。2013年11月高性能计算 Top500 排行中第2名和第6名均为 Cray 机。

- 云计算之父——约翰·麦卡锡（John McCarthy）（见图1.6）

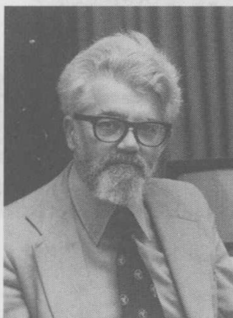


图1.6 约翰·麦卡锡

约翰·麦卡锡1927年生于美国，1951年获得普林斯顿大学数学博士学位。他因在人工智能领域的贡献而在1971年获得图灵奖，麦卡锡真正广为人知的称呼是"人工智能之父"，因为他在1955年的达特矛斯会议上提出了"人工智能"这个概念，使人工智能成为了一门新的学科。1958年发明了 LISP 语言，而 LISP 语言中的 MapReduce 在几十年后成为了 Google 云计算和大数据系统中最为核心的技术。麦卡锡更为富有远见的预言是他在1960年提出的"今后计算机将会作为公共设施提供给公众"这一观点与现在的云计算的理念竟然丝毫不差。正是由于他提前半个多世纪就预言了云计算这种新的模式，因此我们将他称为"云计算之父"。

- 互联网之父——蒂姆·伯纳斯·李（Tim Berners-Lee）（见图1.7）



图1.7 蒂姆·伯纳斯·李

云计算的出现得益于网络的发展，特别是互联的出现大大推动了网络技术的发展，从而使资源和服务能通过网络提供给用户。蒂姆·伯纳斯·李 1955 年生于英国，是英国皇家学会会员，英国皇家工程师学会会员，美国国家科学院院士。1989 年 3 月他正式提出万维网的设想，1990 年 12 月25 日，他在日内瓦的欧洲粒子物理实验室里开发出了世界上第一个网页。最为让人值得尊敬的是他把这一技术免费公开并推广到全世界，这是一个真正科学家的胸怀。

让我们再次访问世界上第一个网页 http://info.cern.ch 以表示向他的敬意。由于他的杰出贡献，他被称为“互联网之父”。

- 大数据之父——吉姆·格雷（Jim Gray）（见图 1.8）

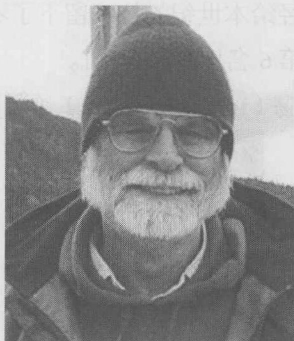


图 1.8 吉姆·格雷

云计算和大数据是密不可分的两个概念，云计算时代网络的高度发展，每个人都成为了数据产生者，物联网的发展更是使数据的产生呈现出随时、随地、自动化、海量化的特征，大数据不可避免地出现在了云计算时代。吉姆·格雷生于 1944 年，在著名的加州大学伯克利分校计算机科学系获得博士学位，是声誉卓著的数据库专家，1998 年度的图灵奖获得者，2007 年 1 月11 日在美国国家研究理事会计算机科学与通信分会上吉姆·格雷明确地阐述了科学研究第四范式，认为依靠对数据分析挖掘也能发现新的知识，这一认识吹响了大数据前进的号角，计算应用于数据的观点在当前的云计算大数据系统中得到了大量的体现。在他发表这一演讲后的十几天，2007 年 1 月 28 号格雷独自架船出海就再也没有了音讯，虽然经多方的努力搜索却没有发现一丝他的信息，人们再也没能见到这位天才的科学家。

3．云计算与大数据的国内发展现状

云计算与大数据概念进入中国以来，国内高度重视云计算产业和技术的发展，中国电子学会率先成立了云计算专业委员会，并在 2009 年举办了第一届中国云计算大会，该委员会在本次大会后，每年都会再举办一次，成为云计算领域的一个重要会议，同时每年出版一本《云计算技术发展报告》，报道当年云计算的发展状况。中国计算机学会于 2012 年成立了大数据专家委员会， 2013 年发布了《中国大数据技术与产业发展白皮书》，并举办了第一届 CCF 大数据学术会议。

国内的研究机构也纷纷开展云计算、大数据研究工作，如清华大学、中国科学院计算所、华中科技大学、成都信息工程学院并行计算实验室都在开展相关的研究工作。科研人员逐步发

现在云计算的新的体系下，有大量需要研究解决的问题，如理论框架、安全机制、调度策略、能耗模型、数据分析、虚拟化、迁移机制等。自“第四范式”提出后，数据成为科学研究的研究对象，大数据概念成为云计算之后信息产业的又一热点，成为科研领域研究的热点。国家自然科学基金反映了我国科研领域的进展，2009~2013 年云计算、大数据、数据中心方向的国家自然科学基金立项数据如图 1.9 所示。

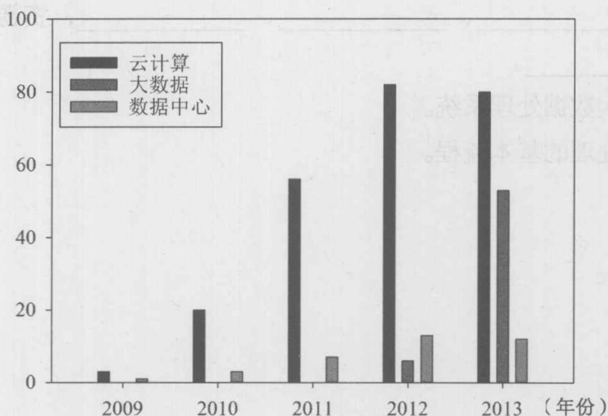


图 1.9　云计算、大数据、数据中心方向的国家自然科学基金立项情况

三个方向在过去的 4 年中经历了迅速发展的过程，云计算从 2008 年开始进入中国，2009 年开始有项目立项，之后云计算立项数目开始快速增长，成为三个方向中立项数目最多的方向；大数据的概念较为新颖，自 2012 开始提出，当年立项 6 项，2013 年这一数字便迅速攀升至 53 项，充分体现大数据在科研领域受到的重视程度；云计算和大数据的发展推动数据中心规模的不断增加，数据中心的建设、运营面临很多新问题，数据中心也成为相关的研究热点。

国内的企业也对云计算、大数据给予了高度关注，华为、中兴、阿里、腾讯都宣布了自己庞大的云计算计划。这些企业多年来积累的数据在大数据时代将发挥巨大作用。数据分析、数据运营的作用已经显现出来，拥有用户数据的 IT 企业对传统的行业产生了巨大影响，“数据为王”的时代正在到来。

练习题

1. 在信息产业的发展历程中，__________、__________作为两个重要的内在动力在不同时期起着重要作用。

2. ________________建造了世界上第一台基于晶体管的超级计算机，被誉为“超级计算机之父”。

3. ________________最早预言了“今后计算机将会作为公共设施提供给公众”，被誉为“云计算之父”。

4. 万维网的发明人、世界上第一个网页的开发者是________________。

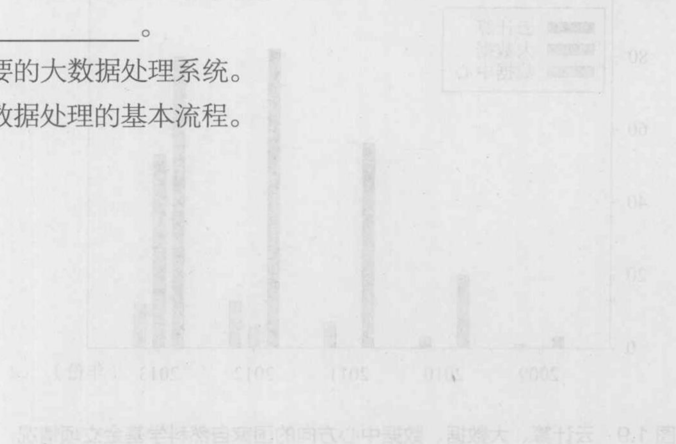
5. ____________提出了第四范式，被誉为“大数据之父”。

6. MapReduce 的思想来源是____________语言。

7. 按照资源封装层次，云计算可分为__________、__________、__________三种类型。

8. 与传统的资源提供方式相比，云计算具有什么特点？

9. 云计算的主要技术路线为资源聚合型和资源切分型，其中资源聚合型的典型系统有__________、__________、__________、__________，资源切分型的典型系统有__________、__________。



10. 简述主要的大数据处理系统。

11. 简述大数据处理的基本流程。

PART 2

第2章 云计算与大数据的相关技术

2.1 云计算与大数据

云计算与大数据相比云计算更像是对一种新的技术模式的描述而不是对某一项技术的描述，而大数据则较为确切地与一些具体的技术相关联。目前新出现的一些技术如 Hadoop、HPCC、Storm 都较为确切地与大数据相关，同时并行计算技术、分布式存储技术、数据挖掘技术这些传统的计算机学科在大数据条件下又再次萌发出生机，并在大数据时代找到了新的研究内容。

大数据其实是对面向数据计算技术中对数据量的一个形象描述，通常也可以被称为海量数据。云计算整合的资源主要是计算和存储资源，云计算技术的发展也清晰地呈现出两大主题——计算和数据。伴随这两大主题，出现了云计算和大数据这两个热门概念，任何概念的出现都不是偶然的，取决于当时的技术发展状况。

李国杰院士认为“信息系统需要从数据围绕着处理器转改为处理能力围绕着数据转，将计算用于数据，而不是将数据用于计算”。海量的数据本身很难直接使用，只有通过处理的数据才能真正地成为有用的数据，因此云计算时代计算和数据两大主题可以进一步明确为数据和针对数据的计算，计算可以使海量的数据成为有用的信息，进而处理成为知识。目前提到云计算时，有时将云存储作为单独的一项技术来对待，只是把网络化的存储笼统地称为云存储，事实上在面向数据的时代不管是出现了云计算的概念还是大数据的概念，存储都不是一个独立存在的系统。特别是在集群条件下，计算和存储都是分布式的，如何让计算“找”到自己需要处理的数据是云计算系统需要具有的核心功能。面向数据要求计算是面向数据的，那么数据的存储方式将会深刻地影响计算实现的方式。这种在分布式系统中实现计算和数据有效融合从而提高数据处理能力，简化分布式程序设计难度，降低系统网络通信压力从而使系统能有效地面对大数据处理的机制称为计算和数据的协作机制，在这种协作机制中计算如何找到数据并启动分布式处理任务的问题是需要重点研究的课题，在本文中这一问题被称为计算和数据的位置一致性问题。

面向数据也可以更准确地称为“面向数据的计算”，面向数据要求系统的设计和架构是围绕数据为核心展开的，面向数据也是云计算系统的一个基本特征，而计算与数据的有效协作是面向数据的核心要求。

回顾计算机技术的发展历程，可以清晰地看到计算机技术从面向计算逐步转变到面向数据的过程。从面向计算到面向数据是技术发展的必然趋势，并不能把云计算的出现归功于任何的个人和企业。这一过程的描述如图 2.1 所示，该图从硬件、网络和云计算的演进过程等方面以时间为顺序进行了纵向和横向的对比。

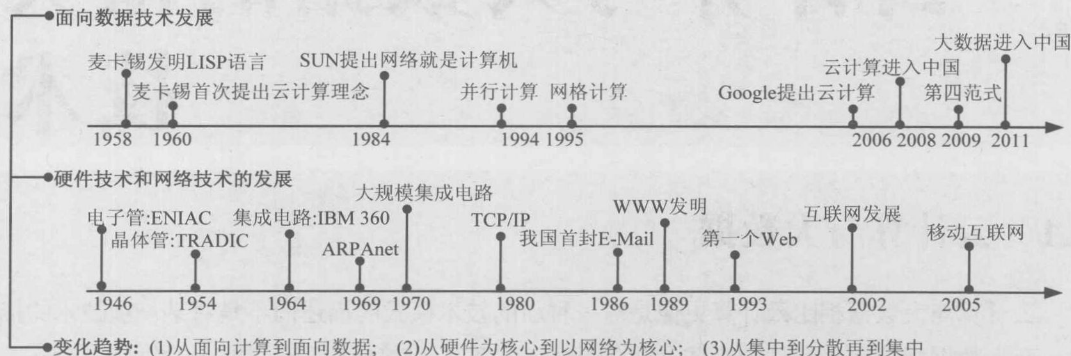


图 2.1　计算机技术向云计算的演进

从图 2.1 中可以看出，在计算机技术的早期由于硬件设备体积庞大、价格昂贵，这一阶段数据的产生还是“个别”人的工作。这个时期的数据生产者主要是科学家或军事部门，他们更关注计算机的计算能力，计算能力的高低决定了研究能力和一个国家军事能力的高低。相对而言由于这时数据量很小，数据在整个计算系统中的重要性并不突出。这时网络还没有出现，推动计算技术发展的主要动力是硬件的发展，这个时期是硬件的高速变革时期，硬件从电子管迅速发展到大规模集成电路。1969 年 ARPANET 的出现改变了整个计算机技术的发展历史，网络逐步成为推动技术发展的一个重要力量，1989 年 Tim Berners-Lee 发明的万维网改变了信息的交流方式，特别是高速移动通信网络技术的发展和成熟使现在数据的生产成为全球人的共同活动，人们生产数据不再是在固定时间和固定地点进行，而是随时随地都在产生数据。微博、博客、社交网、视频共享网站、即时通信等媒介随时都在生产着数据并被融入全球网络中。

从云计算之父 John McCarthy 提出云计算的概念到大数据之父 Gray 等人提出科学研究的第四范式，时间已经跨越了半个世纪。以硬件为核心的时代也是面向计算的时代，那时数据的构成非常简单，数据之间基本没有关联性，物理学家只处理物理实验数据，生物学家只处理生物学数据，计算和数据之间的对应关系是非常简单和直接，这个时期研究计算和存储的协作机制并没有太大的实用价值。到了以网络为核心的时代数据的构成变得非常复杂，数据来源多样化，不同数据之间存在大量的隐含关联性，这时计算所面对的数据变得非常复杂，如社会感知、微关系等应用将数据和复杂的人类社会运行相关联，由于人人都是数据的生产者，人们之间的社会关系和结构就被隐含到了所产生的数据之中。数据的产生目前呈现出了：大众化、自动化、连续化、复杂化的趋势。云计算、大数据概念正是在这样的一个背景下出现的。这一时期的典型特征就是计算必须面向数据，数据是架构整个系统的核心要素，这就使计算和存储的协作机制研究成为需要重点关注的核心技术，计算能有效找到自己需要处理的数据，可以使系统能更

高效地完成海量数据的处理和分析。云计算和大数据这两个名词也可看作是描述了面向计算时代信息技术的两个方面，云计算侧重于描述资源和应用的网络化交付方法，大数据侧重于描述面向数据时代由于数据量巨大所带来的技术挑战。

信息技术领域提出的面向数据的概念同时也开始深刻地改变了科学研究的模式，2007 年著名的数据库专家 Gray 提出了科学研究的第四范式。他认为利用海量的数据可以为科学研究和知识发现提供除经验、理论、计算外的第四种重要方法。科学研究的四个范式的发展历程也同样反映了从面向计算走向面向数据的过程。

如图 2.2 所示，人类早期知识的发现主要依赖于经验、观察和实验，需要的计算和产生的数据都是很少的。人类在这一时期对于宇宙的认识都是这样形成的，就像伽利略为了证明自由落体定理，是通过在比萨斜塔扔下两个大小不一的小球一样，人类在那个时代知识的获取方式是原始而朴素的。当人类知识积累到一定的程度后，知识逐渐形成了理论体系，如牛顿力学体系、Maxwell 的电磁场理论，人类可以利用这些理论体系去预测自然并获取新的知识，这时对计算和数据的需求已经在萌生，人类已可以依赖这些理论发现新的行星，如海王星、冥王星的发现不是通过观测而是通过计算得到。计算机的出现为人类发现新的知识提供了重要的工具。这个时代正好对应于面向计算的时代，可以在某些具有完善理论体系领域利用计算机仿真计算来进行研究。这时计算机的作用主要是计算，例如人类利用仿真计算可以实现模拟核爆这样的复杂计算。现在人类在一年内所产生的数据可能已经超过人类过去几千年产生的数据的总和，即使是复杂度为 $O(n)$ 的数据处理方法在面对庞大的 n 时都显得力不从心，人类逐步进入面向数据的时代。第四范式说明可以利用海量数据加上高速计算发现新的知识，计算和数据的关系在面向数据时代变得十分紧密，也使计算和数据的协作问题面临巨大的技术挑战。

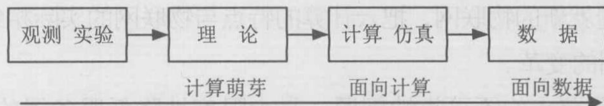


图 2.2　科学研究四个范式的发展历程

2.2　云计算与物联网

云计算和物联网的出现时间非常接近，以至于有一段时间云计算和物联网两个名词总是同时出现在各类媒体上。物联网的出现部分得益于网络的发展，大量传感器数据的收集需要良好的网络环境，特别是部分图像数据的传输更是对网络的性能有较高的要求。在物联网技术中传感器的大量使用使数据的生产实现自动化，数据生产的自动化也是推动当前大数据技术发展的动力之一。物联网的英文名称为“The Internet of Things”，简称：IOT。由该名称可见，物联

网就是“物物相连的互联网”。这有两层意思：第一，物联网的核心和基础仍然是互联网，是在互联网基础之上的延伸和扩展的一种网络；第二，其用户端延伸和扩展到了任何物品与物品之间，进行信息交换和通信。因此，物联网的定义是通过射频识别(RFID)装置、红外感应器、全球定位系统、激光扫描器等信息传感设备，按约定的协议，把任何物品与互联网相连接，进行信息交换和通信，以实现智能化识别、定位、跟踪、监控和管理的一种网络。明确的物联网概念最早是由美国麻省理工大学 Auto-ID 实验室在 1999 年提出的，最初是为了提高基于互联网流通领域信息化水平而设计的。物联网这个概念可以认为对一类应用的称呼，物联网与云计算技术的关系从定义上讲应用与平台的关系。

物联网系统需要大量的存储资源来保存数据，同时也需要计算资源来处理和分析数据，当前我们所指的物联网传感器连接呈现出以下的特点：

- 连接传感器种类多样；
- 连接的传感器数量众多；
- 连接的传感器地域广大。

这些特点都会导致物联网系统会在运行过程中产生大量的数据，物联网的出现使数据的产生实现自动化，大量的传感器数据不断地在各个监控点产生，特别是现在信息采样的空间密度和时间密度不断增加，视频信息的大量使用，这些因素也是目前导致大数据概念出现的原因之一。

物联网的产业链可以细分为标识、感知、处理和信息传送 4 个环节，每个环节的关键技术分别为 RFID、传感器、智能芯片和电信运营商的无线传输网络。云计算的出现使物联网在互联网基础之上延伸和发展成为可能。物联网中的物，在云计算模式中，它相当于是带上传感器的云终端，与上网本、手机等终端功能相同。这也是物联网在云计算日渐成熟的今天，才能重新被激活的原因之一。

新的平台必定造就新的物联网，把云计算的特点与物联网的实际相结合，云计算技术将给物联网带来以下深刻的变革。

（1）解决服务器节点的不可靠性问题，最大限度地降低服务器的出错率。近年来，随着物联网从局域网走向城域网，其感知信息也呈指数型增长，同时导致服务器端的服务器数目呈线性增长。服务器数目多了，节点出错的概率肯定也随之变大，更何况服务器并不便宜。如今商场如战场，节点不可信问题使得一般的中小型公司要想独自撑起一片属于自己的天空，那是难上加难。

而在云计算模式中，因为“云”有成千上万、甚至上百万台服务器，即使同时宕掉几台，“云”中的服务器也可以在很短的时间内，利用冗余备份、热拔插、RAID 等技术快速恢复服务。

例如，Google 公司不再是一味地追求单个服务器的性能参数，而是更多地关注如何用堆积如山的集群来弥补单个服务器的性能不足。在对单个服务器性能要求的降低的同时也减少了相应的资金需求。至于对于宕机的服务器，Google 采用的是直接换掉。云计算集群的加入，能够保证物联网真正实现无间断的安全服务。

（2）低成本的投入可以换来高收益，让限制访问服务器次数的瓶颈成为历史。服务器相关

硬件资源的承受能力都是有一定范围的，当服务器同时响应的数量超过自身的限制时，服务器就会崩溃。而随着物联网领域的逐步扩大，物的数量呈几何级增长，而物的信息也呈爆炸性增长，随之而来的访问量空前高涨。

因此，为了让服务器能安全可靠地运行，只有不断增加服务器的数量和购买更高级的服务器，或者限制同时访问服务器的数量。然而这两种方法都存在致命的缺点：服务器的增加，虽能通过大量的经费投入解决一时的访问压力，但设备的浪费却是巨大的。而采用云计算技术，可以动态地增加或减少云模式中服务器的数量和提高质量，这样做不仅可以解决访问的压力，还经济实惠。

（3）让物联网从局域网走向城域网甚至是广域网，在更广的范围内进行信息资源共享。局域网中的物联网就像是一个超市，物联网中的物就是超市中的商品，商品离开这个超市到另外的超市，尽管它还存在，但服务器端内该物体的信息会随着它的离开而消失。其信息共享的局限性不言而喻。

但通过云计算技术，物联网的信息直接存放在 Internet 的“云”上，而每个“云”有几百万台服务器分布在全国甚至是全球的各个角落，无论这个物走到哪儿，只要具备传感器芯片，“云”中最近的服务器就能收到它的信息，并对其信息进行定位、分析、存储、更新。用户的地理位置也不再受限制，只要通过 Internet 就能共享物体的最新信息。

（4）将云计算与数据挖掘技术相结合，增强物联网的数据处理能力，快速做出商业抉择。伴随着物联网应用的不断扩大，业务应用范围从单一领域发展到所有的各行各业，信息处理方式从分散到集中，产生了大量的业务数据。

运用云计算技术，由云模式下的几百万台的计算机集群提供强大的计算能力，并通过庞大的计算机处理程序自动将任务分解成若干个较小的子任务，快速地对海量业务数据进行分析、处理、存储、挖掘，在短时间内提取出有价值的信息，为物联网的商业决策服务。这也是将云计算技术与数据挖掘技术相结合给物联网带来的一大竞争优势。

任何技术从萌芽到成型，再到成熟，都需要经历一个过程。云计算技术作为一项有着广泛应用前景的新兴前沿技术，尚处于成型阶段，自然也面临着一些问题。

首先是标准化问题。虽然云平台解决的问题一样，架构一样，但基于不同的技术、应用，其细节很可能完全不同，从而导致平台与平台之间可能无法互通。目前在 Google、EMC、Amazon 等云平台上都存在许多云技术打造的应用程序，却无法跨平台运行。这样一来，物联网的网与网之间的局限性依旧存在。

其次是安全问题。物联网从专用网到互联网，虽然信息分析、处理得到了质的提升，但同时网络安全性也遇到了前所未有的挑战。Internet 上的各种病毒、木马以及恶意入侵程序让架于云计算平台上的物联网处于非常尴尬的境地。

云计算作为互联网全球统一化的必然趋势，其统一虚拟的基础设施平台，方便透明的上层调用接口，计算信息的资源共享等特点，完全是在充分考虑了各行各业的整合需求下才形成的拯救互联网的诺亚方舟。尽管，目前云计算的应用还处在探索测试阶段，但随着物联网界对云计算技术的关注以及云计算技术的日趋成熟，云计算技术在物联网中的广泛应用指日可待。

2.3 一致性哈希算法

2.3.1 一致性哈希算法的基本原理

主从结构的云计算系统负载的均衡往往通过主节点来完成，而一些对等结构的云计算系统可以采用一致性哈希算法来实现负载的均衡，这种模式避免了主从结构云计算系统对主节点失效的敏感。

哈希算法是一种从稀疏值范围到紧密值范围的映射方法，在存储和计算定位时可以被看作是一种路由算法，通过这种路由算法文件块能被唯一地定位到一个节点的位置。传统的哈希算法的容错性和扩展性都不好，无法有效地适应面向数据系统节点的动态变化。1997 年 David Karger 提出了一致性哈希算法来定位数据，实现了云计算系统在节点变化时的单调性，实现了较小的数据迁移代价。Amazon 的云存储系统 Dynamo 改进了基本的一致性哈希算法，引入了虚拟节点，使系统具有更加均衡地存储定位能力。Facebook 开发的 Cassandra 系统也是采用了一致性哈希算法的存储管理算法。一致性哈希算法及其改进算法已成为分布式存储领域的一个标准技术。使用一致性哈希算法的系统无需中心节点来维护元数据，解决了元数据服务器的单点失效和性能瓶颈问题，但对于系统的负载均衡和调度节点的有效性提出了更高的要求。

传统的哈希算法在节点数没有变化时能很好地实现数据的分配，但当节点数发生变化时传统的哈希算法将对数据进行重新的分配，这样系统恢复的代价就非常大。例如系统中节点数为 N，传统的哈希算法的计算方法为 HASH（Key）% N，当 N 发生变化时整个哈希的分配次序将完全重新生成。云计算系统通常涉及大量的节点，节点的失效和加入都是非常常见的，传统的哈希算法无法满足这种节点数目频繁变化的要求。

一致性哈希的设计目标就是解决节点频繁变化时的任务分配问题，一致性哈希将整个哈希值空间组织成一个虚拟圆环（如图 2.3 所示），假设某哈希函数 H 的值空间为 $0\sim(2^{32}-1)$，即 32 位无符号整数，将各节点用 H 函数哈希，可以将服务器的 IP 或主机名作为关键字哈希，这样每个节点就能确定其在哈希环上的位置，将 Key 用 H 函数映射到哈希空间的一个值，沿该值向后，将遇到的第一个节点作为处理节点，若某个 Key 的 HASH 值落在 node1 和 node2 各自 HASH 值的中间位置，则此 Key 对应的业务请求由 node2 处理。当增加服务节点时，只会影响与之相邻的某一节点，其他节点不受影响。如果在 node2 和 node4 之间增加一个 node5，则只有 node4 处理的部分 Key（HASH 值落在 node2 之后、node5 之前的那部分 Key）变为由 node5 来处理，其他节点处理的 Key 不变。如果节点数不多，则将这些节点映射到值空间之后，分布可能会很不均匀，必然会造成个别节点处理的 Key 数量远大于其他节点，这就起不到负载均衡的效果。这可以通过引入虚拟节点的方式解决，即对每一个节点计算多个 HASH 值，尽量保证这些 HASH 值比较均匀地分布在值空间中。当根据 Key 查找节点时，找到的是虚拟节点，然后再根据虚拟节点查找对应的真实节点。

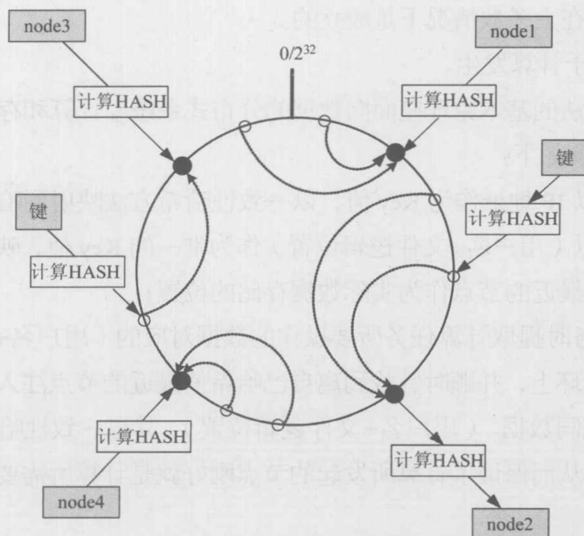


图 2.3 一致性哈希原理

简单来说，一致性哈希算法的基本实现过程为：对 Key 值首先用 MD5 算法将其变换为一个长度 32 位的十六进制数值，再用这个数值对 2^{32} 取模，将其映射到由 2^{32} 个值构成的环状哈希空间，对节点也以相同的方法映射到环状哈希空间，最后 Key 值会在环状哈希空间中找到大于它的最小的节点值作为路由值。

一致性哈希算法的采用使集群系统在进行任务划分时不再依赖某些管理节点来维护，并且在节点数据发生变化时能够以最小的代价实现任务的再分配，这一优点特别符合云计算系统资源池弹性化的要求，因此一致性哈希算法成为了实现无主节点对等结构集群的一种标准算法。

2.3.2 一致性哈希算法中计算和存储位置的一致性

基于一致性哈希的原理可以给出计算和存储的一致性哈希方法，从而使计算能在数据存储节点发起。对于多用户分布式存储系统来说："用户名+逻辑存储位置"所构成的字符串在系统中是惟一确定的，如属于用户 wang，逻辑存储位置为/test/test1.txt 的文件所构成的字符串"wang/test/test1.txt"在系统中一定是惟一的，同时某一个计算任务需要对 test1.txt 这个文件进行操作和处理，则它一定会在程序中指定用户名和逻辑位置，因此存储和计算 test1.txt 都利用相同的一致性哈希算法就能保证计算被分配的节点和当时存储 test1.txt 文件时被分配的节点是同一个节点。

现在以下面这个应用场景为例，说明一致性哈希算法实现计算和存储位置一致性的方法：

（1）面向相对"小"数据进行处理，典型的文件大小为 100MB 之内，通常不涉及对文件的分块问题，这一点与 MapReduce 框架不同；

（2）待处理数据之间没有强的关联性，数据块之间的处理是独立的，数据处理是不需要进行数据块之间的消息通信，保证节点间发起的计算是低耦合的计算任务；

（3）程序片的典型大小远小于需要处理的数据大小，计算程序片本质上也可以看作是一种

特殊的数据，这一假设在大多数情况下是成立的。

（4）数据的存储先于计算发生。

根据一致性哈希算法的基本原理在面向数据的分布式系统中计算和存储位置一致性方法如图 2.4 所示，其主要步骤如下：

① 将服务器节点以 IP 地址作为 Key 值，以一致性哈希方法映射到哈希环上；

② 在数据存储时以（用户名+文件逻辑位置）作为唯一的 Key 值，映射到哈希环上，并顺时针找到离自己哈希值最近的节点作为实际数据存储的位置；

③ 在发起计算任务时提取计算任务所要操作的数据对应的（用户名+文件逻辑位置）值作为 Key 值，映射到哈希环上，并顺时针找到离自己哈希值最近的节点注入程序并发起计算的节点。由于相同用户的相同数据（用户名+文件逻辑位置）其在一致性哈希算法作用下一定会被分配到相同的节点，从而保证了计算所发起的节点刚好就是计算所需要处理的数据所在的节点。

在这种算法的支持下只要计算程序片需要处理的数据逻辑位置是确定的，系统就会将计算程序片路由到数据存储位置所在的节点，这时节点间的负载均衡性是由数据分布的均衡化来实现的。

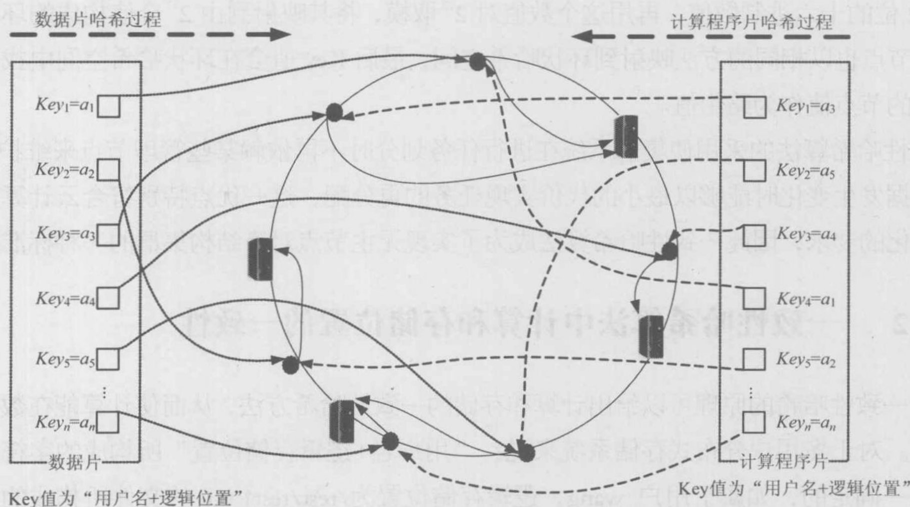


图 2.4　一致性哈希算法实现计算与数据的位置一致性

一致性哈希算法可以实现无中心节点的计算和数据定位，使计算可以惟一地找到其所要处理和分析的数据，使计算能最大可能地在数据存储的位置发起，从而节约大量的网络资源，同时避免了系统单点失效造成的不良影响。利用一致性哈希方法在面对海量文件时系统不用维护一个庞大的元数据库用于保存文件的存储信息，计算寻找数据的速度非常直接，路由的算法复杂度非常低。需要存储大量 Key-Value 的 Amazon 的电子商务应用和 Facebook 的社交网站应用都采用了一致性哈希算法。

2.4 非关系型数据库

2.4.1 从关系型数据库到非关系型数据库

关系型数据库（Relational Database）技术是1970年埃德加·科德（Edgar Frank Codd）所提出的，关系型数据库克服了网络数据库模型和层次数据库模型的一些弱点。1981年埃德加·科德因在关系型数据库方面的贡献获得了图灵奖，因此埃德加·科德也被称为"关系数据库之父"（见图 2.5）。关系型数据库几十年来一直是统治数据库技术的核心标准，目前主要的数据库系统仍然采用的是关系型数据库。埃德加·科德发明的关系数据库不仅有一个坚实的数学基础，即关系代数，而且埃德加·科德从关系代数的基础推演出一套关系数据库的理论。这个理论包括一系列"范式"，可以用来检查数据库是否有冗余性和不一致等性质。另外，埃德加·科德也在关系代数基础之上定义了一系列通用的数据基本操作。但云计算和大数据技术的出现逐步动摇了关系型数据库的统治地位。

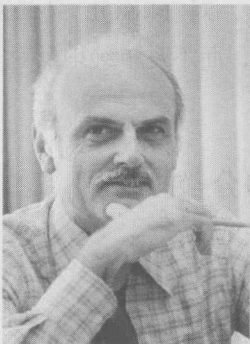


图 2.5 关系数据库之父—— Edgar Frank Codd

随着信息产业的发展，特别是在云计算和互联网 Web 2.0 蓬勃发展的今天，数据库系统成为了 IT 架构中信息存储和处理的必要组成部分，Web 2.0 是相对 Web 1.0 的新的一类互联网应用的统称。Web 1.0 的主要特点在于用户通过浏览器获取信息。Web 2.0 则更注重用户的交互作用，用户既是网站内容的浏览者，也是网站内容的制造者。所谓网站内容的制造者是说互联网上的每一个用户不再仅仅是互联网的读者，同时也成为互联网的作者；不再仅仅是在互联网上冲浪，同时也成为波浪制造者；在模式上由单纯的"读"向"写"以及"共同建设"发展；由被动地接收互联网信息向主动创造互联网信息发展，从而更加人性化。云计算资源网络化的提供方式更是为 Web 2.0 发展提供了无限的想象空间，从这一点看我们已很难将这两者完全区分开来。云计算技术对数据库高并发读/写的需求，对海量数据的高效率存储和访问的需求，对数据库的高可扩展性和高可用性的需求都让传统关系型数据库系统显得力不从心。同时关系型数据库技术中的一些核心技术要求如：数据库事务一致性需求，数据库的写实时性和读实时性

需求，对复杂的 SQL 查询，特别是多表关联查询的需求等在 Web 2.0 技术中却并不是必要的，而且系统为此付出了较大的代价。

关系型数据库技术的出现是云计算、大数据技术的必然需求，非关系型数据库可以被称为一项数据库的革命，从 2009 年开始，在云计算的发展和开源社区的推动下，非关系型数据库的发展显示了较强的活力，也得到了越来越多的用户关注和认可。目前已经有多家大型 IT 企业已经采用非关系型数据库作为重要的生产系统基础支撑，比如 Google 的 BigTable，Amazon 的 Dynamo，以及 Digg、Twitter、Facebook 在使用的 Cassandra 等。

2.4.2 非关系型数据库的定义

非关系型数据库，又被称为 NoSQL(Not Only SQL)，意为不仅仅是 SQL(Structured Query Language，结构化查询语言)，据维基百科介绍，NoSQL 最早出现于 1998 年，是由 Carlo Strozzi 最早开发的一个轻量、开源、不兼容 SQL 功能的关系型数据库。2009 年，在一次关于分布式开源数据库的讨论会上，再次提出了 NoSQL 的概念，此时 NoSQL 主要是指非关系型、分布式、不提供 ACID（数据库事务处理的四个基本要素）的数据库设计模式。同年，在亚特兰大举行的“NO:SQL(east)”讨论会上，对 NoSQL 最普遍的定义是“非关联型的”，强调 Key-Value 存储和文档数据库的优点，而不是单纯地反对 RDBMS，至此，NoSQL 开始正式出现在世人面前。

2.4.3 非关系型数据库的分类

NoSQL 描述的是大量结构化数据存储方法的集合，根据结构化方法以及应用场合的不同，主要可以将 NoSQL 分为以下几类。

（1）Column-Oriented。

面向检索的列式存储，其存储结构为列式结构，不同于关系型数据库的行式结构，这种结构会让很多统计聚合操作更简单方便，使系统具有较高的可扩展性。这类数据库还可以适应海量数据的增加以及数据结构的变化，这个特点与云计算所需的相关需求是相符合的。比如 Google Appengine 的 Big Table 以及相同设计理念的 Hadoop 子系统 HaBase 就是这类的典型代表。需要特别指出的是，BigTable 特别适用于 MapReduce 处理，这对于云计算的发展有很高的适应性。

（2）Key-Value。

面向高性能并发读/写的缓存存储，其结构类似于数据结构中的 Hash 表，每个 Key 分别对应一个 Value，能够提供非常快的查询速度、大数据存放量和高并发操作，非常适合通过主键对数据进行查询和修改等操作。Key-Value 数据库的主要特点是具有极高的并发读/写性能，非常适应作为缓存系统使用。MemcacheDB、Berkeley DB、Redis、Flare 就是 Key-Value 数据库的代表。

（3）Document-Oriented。

面向海量数据访问的文档存储，这类存储的结构与 Key-Value 非常相似，也是每个 Key 分别对应一个 Value，但是这个 Value 主要以 JSON（JavaScript Object Notations）或者 XML 等格式的文档来进行存储。这种存储方式可以很方便地被面向对象的语言所使用。这类数据库可以在海量的数据中快速查询数据，典型代表为 MongoDB、CouchDB 等。

NoSQL 具有扩展简单、高并发、高稳定性、成本低廉等优势，也存在一些问题。例如，NoSQL 暂不提供对 SQL 的支持，会造成开发人员的额外学习成本；NoSQL 大多为开源软件，其成熟度与商用的关系型数据库系统相比有差距；NoSql 的架构特性决定了其很难保证数据的完整性，适合在一些特殊的应用场景使用。

2.5 集群高速通信标准 InfiniBand

集群结构是云计算系统的基本结构之一，在集群结构中节点之间的协调和数据传送一般都通过消息传递机制进行，在传统的高性能计算系统集群内部的网络通信能力成为了影响集群计算能力的一个重要因素，由于受到网络通信速度的制约甚至提出以计算换通信的编程理念。云计算系统中集群的规模变得空前巨大，为了很好地实现集群内部的调度和协调高速的网络通信是必不可少的。InfiniBand 就是目前较为常见的一种高速集群通信协议，它在高性能计算领域已得到广泛的应用。

InfiniBand 是一种全新、功能强大、设计用来支持 Internet 基础设施 I/O 互联的体系结构。InfiniBand 被工业界的顶级公司所支持，执行委员会成员包括：Compaq、Dell、Hewlett Packard、IBM、Intel、Microsoft 和 Sun。InfiniBand 行业协会成员总计超过 220 个。InfiniBand 被主要的 OEM 服务器生产商所支持，用来作为下一代服务器的 I/O 互联标准，是第一个高性能的机箱内部 I/O 互联方式得到延伸的工业标准。InfiniBand 是惟一既提供机箱内底板互联解决方案，又可以使高速带宽延伸出机箱外部的互联标准，是一种把 I/O 和系统域网络统一起来的标准。与其他商品化系统域网络不同，InfiniBand 网络首先由行业协会制定标准，然后生产商根据标准制造出 InfiniBand 设备，不同生产商的产品要求具有互操作性，这样可以使 InfiniBand 产品具有更好的性能价格比。

InfiniBand 是在 1999 年由 Future IO 和 NGIO 两个标准整合而来的新型互联技术。它将复杂的 I/O 系统与 CPU、内存分开，使 I/O 子系统独立，采用基于包交换的高速串行链路和可扩展的高速交换网络替代共享总线结构，提供了高带宽、低延迟、可扩展的 I/O 互联。基于 I/O 通路共享机制的 InfiniBand 提供了一种连接计算机的新途径，采用 InfiniBand 之后，I/O 不再是服务器的组成部分，这时远程存储设备、I/O 设备和服务器之间的互联是通过 InfiniBand 交换机和路由器完成的，克服了传统的共享 I/O 总线结构的种种弊端。InfiniBand 具有传输速率高、传输距离长、接口功耗低以及噪声容限高等特性，单线传输速率为 2.5Gbps，可通过 2、4 或 12 线并行来扩展通道带宽，峰值带宽高达 2.5、10、30Gbps（1x、4x、12x 线）。

InfinBand 既可作为系统内部互联技术又可作为网络互联技术，既可用于构造高性能刀片服务器（Blade Server），又可用于构建具有高可用性和高可伸缩性的大规模集群系统，还可用于构建性能卓越的存储区域网络 SAN。

2.6　云计算大数据集群的自组织特性

自组织理论产生于 20 世纪七八十年代，主要研究混沌系统在随机识别时系统内部自发地由无序变为有序、形成耗散结构的过程，被物理、化学、生物学、社会学、心理学等领域广泛研究，其中一些研究者将自组织理论的思想、模型与计算机科学、控制理论等学科相结合，形成了一些新兴的研究方向。自组织算法按照自下而上的机制进行控制，与一般采用中心控制算法不同，在进化计算、任务分配、网络自组织演化等中得到广泛的研究。典型的进化算法（如蚁群算法）由个体依据简单规则，通过正反馈、分布式协作依靠群体的力量自动寻找最优路径；任务分配通过多智能体并行实现在离散、有限的数据结构上，寻找一个满足给定约束条件并使目标函数值达到最大或最小的解；人工生命进化由人工分子的非线性相互作用引起，是远离热平衡的相变，是自组织的研究方向之一；网络自组织演化研究在内在机制驱动下，网络自行从简单向复杂、从粗糙向细致方向发展，不断地以局部优化达到全局优化的过程。

研究自组织必然要提到耗散结构理论，比利时科学家普里高津 1969 年提出耗散结构理论后，这一理论就被广泛地应用于物理、生物、化学乃至社会科学的研究中。由耗散结构引出的自组织现象更是打破了人类对自然的常规认识。耗散结构理论和协同学从宏观、微观及两者的联系上回答了系统自动走向有序结构的问题，其成果被称为自组织理论。

出现自组织现象的系统必须是耗散结构的。耗散结构理论指出：一个开放系统处在远离平衡态的非线性区域，当系统的某个参数变化到一定的临界值时，通过涨落，系统发生突变，即非平衡相变，其状态可能从原来的混乱无序的状态转变到一种在时间、空间上或功能上有序的新状态，这种新的有序结构（耗散结构）需要系统不断地与外界交换物质和能量才能得以维持并保持一定的稳定性。从热力学的观点来说，"自组织"是指一个系统通过与外界交换物质、能量和信息，而不断地降低自身的熵含量，提高其有序度的过程。

耗散结构理论指出一个系统要形成耗散结构需要满足以下几个条件：

（1）系统必须是开放的；

（2）系统需要保持远离平衡态；

（3）系统内部存在着非线性相互作用；

（4）系统存在涨落。

云计算大数据集群具有数量众多的节点，对比耗散结构的特点，集群系统本身满足耗散结构的要求：

（1）云计算大数据集群系统存在着与系统外的能量、物质和信息的交换，满足耗散结构的第一个条件；

（2）一个运营良好的云计算大数据集群系统节点负载是被有效均衡的，系统的均衡状态是一种熵值较低的远离平衡态的状态，满足耗散结构的第二个条件；

（3）云计算大数据集群系统是一种具有海量节点的高耦合系统，由于在云计算系统的节点间中存在着频繁的计算迁移、存储迁移、计算备份、存储备份、节点失效处理等节点间的高耦合性操作，系统内部存在着大量的相互作用，系统内部存在着非线性相互作用，满足耗散结构的第三个条件；

（4）云计算大数据集群系统内的各种状态参数会随着外部请求的变化出现涨落，满足耗散结构的第四个条件。

云计算大数据集群系统的抽象描述如图 2.6 所示。

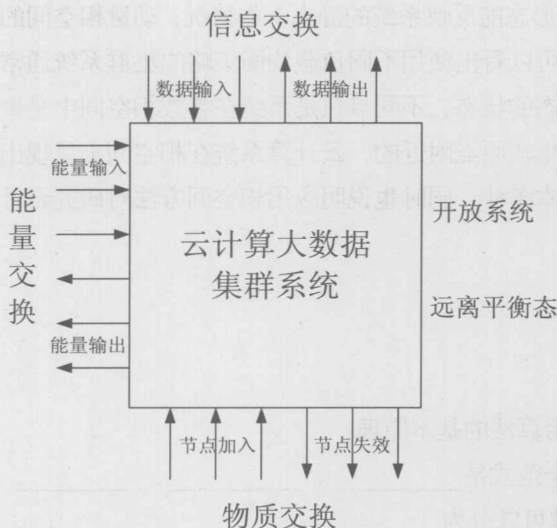


图 2.6　云计算大数据集群系统的抽象描述

从图 2.6 中我们可以看出：集群系统运行时需要能量的输入，同时系统内的一部分能量又会转变为热能流出系统；集群系统需要不断地接受系统外传入的数据并进行处理，同时又会不断地向系统外传出数据；集群系统由于具有很好的弹性规模且又非常大，所以节点的增加和节点的失效都十分频繁；同时由于系统内部又存在大数据迁移、计算迁移、负载均衡等相互作用，因此可以把集群系统抽象地描述为与外界有大量能量、信息、物质交换的开放系统。

图 2.7 所示为 1000 个节点的系统在不同的负载均衡策略下在参数相空间和动量相空间（相空间的基本知识见本书 10.3 节）中的仿真实现结果，体现了集群系统的简单自组织现象，图中上面一排是在参数相空间的仿真结果，下面一排是动量相空间的仿真结果。

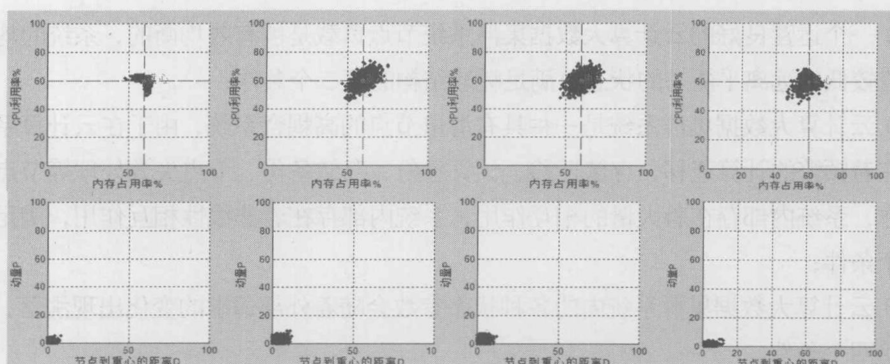


图 2.7 集群系统在不同的负载均衡策略下的简单自组织特性

通常参数相空间的形态能反映系统的静态工作情况，动量相空间的形态能反映系统的动态工作情况，从图 2.7 中可以看出使用不同负载均衡策略的集群系统通常在相空间中都处于广义熵相对较小的远离平衡态的状态，不同点只是系统在参数相空间中是聚集在广义重心位置的，而在动量相空间中是聚集在原点附近的，云计算系统在相空间上呈现出的远离平衡态形态表明其具备自组织出现的基本条件，同时也说明应用相空间方法对研究云计算系统的自组织特性是有效的。

练习题

1. 简述一致性哈希算法的基本原理。
2. 科学研究的四个范式是__________、__________、__________、__________。
3. 物联网的产业链可以分为__________、__________、__________、__________4 个环节。
4. 非关系型数据库可分为__________、__________、__________。

PART 3

第 3 章 虚拟化技术

虚拟化技术和并行计算、分布式计算、网格计算等的发展促进了云计算技术的产生和发展，通过云计算技术，我们将大量的计算机资源组成资源池来创建高度虚拟化的资源提供给用户，即云计算技术解决方案依靠并利用虚拟化提供服务。虚拟化技术主要应用在基础设施即服务的服务模式（IaaS）中，大多资源都可以通过虚拟化技术对其进行统一管理。

虚拟化在计算机领域的发展至今已有 50 多年了，在这期间产生了很多种虚拟化形式，如网络虚拟化、微处理器虚拟化、桌面虚拟化等。这些虚拟化技术的产生、成熟离不开计算机技术的发展。虚拟化从概念上来说就是将在实际环境运行的程序、组件，放在虚拟的环境中来运行，从而达到以小的成本来实现与真实环境相同或类似的功能的目的。

3.1 虚拟化技术简介

3.1.1 虚拟化技术的发展

1．广义上的虚拟化技术

从广义来说，虚拟化技术的范围非常宽，它不但是云计算的核心技术，也是计算机科学的核心技术，甚至是整个工业领域的核心技术。

我们在开汽车的时候，其实就用到了机械的虚拟化技术。汽车机械运动的复杂形态，被逻辑上简化为方向盘、油门、刹车、离合器的简单运动方式。这种简化使汽车机械系统的复杂性被全面屏蔽，转向盘、油门、制动器、离合器成为了最终的人机交互设备。

不只在工业领域，就是在金融领域，也有虚拟化技术的身影，如复杂的股市变化规律被几个简单的指数所描述，指数的变化反映了股票市场的运行情况，甚至整个经济的运行规律。

物理学中也广泛地应用了虚拟化技术，原子核内部的复杂运动情况通过各种宏观的物理设备，如拉曼谱、正电子谱、核磁共振等表现出来，其实我们谁也没有真正地看到过原子核的形态，但我们能通过对这些物理量的分析而得到原子核的结构，这些物理设备就实现了对原子核内部结构的虚拟化工作。

从以上的描述中我们可以看出，虚拟化技术是一门应用很广泛的技术，甚至可以说是一门相当基础的学科。广义地定义虚拟化技术可以这么来看：虚拟化技术就是一种逻辑简化技术，

实现物理层向逻辑层的变化。

从这个定义来看，一个系统采用虚拟化技术后，其对外表现出的运动方式是一种逻辑化的运动方式，而不是真实的物理运动方式。所以采用虚拟化技术能实现对物理层运动复杂性的屏蔽，使系统对外运行状态呈现出简单的逻辑运行状态。

2．计算机学科中的虚拟化技术

早期的计算机大多用于科学计算，计算机不仅价格昂贵，而且硬件资源的利用率低，用户的体验效果也差强人意，从而有了分时系统的提出。为了满足分时系统的需求，克里斯托弗（Christopher Strachey）提出了虚拟化的概念。在 1959 年召开的国际信息处理大会上，其发表了一篇名为《大型高速计算机中的时间共享》（Time Sharing in Large Fast Computers）的学术报告，在这篇文章中他提出了虚拟化的基本概念。虚拟化作为一个概念被正式提出就是从此时开始的。

在随后的 10 年中，由于当时工业、科技条件的限制，计算机的硬件资源是相当昂贵的，IBM 在 1956 年推出的首部磁盘储存器件，总容量仅 5MB，但是平均每 MB 需花费 1 万美元。这远远超出了普通大众的承受范围，严重阻碍了人们对计算机的购买力。为了使昂贵的硬件资源得到充分利用，来提高自己的销售额，IBM 最早发明了一种操作系统虚拟机技术，能够让用户在一台主机上运行多个操作系统，IBM 7044 计算机就是典型的代表。随后虚拟化技术一直只在大型机上应用，而在 PC、服务器的 x86 平台上仍然进展缓慢。

随着科技水平的提高，计算机硬件资源的价格逐渐降低，从 20 世纪 90 年代末开始，x86 计算机由于其成本低廉渐渐代替大型机，为了抢占市场的份额，VMware 就在考虑如何节省客户的开支，来提高自己产品的竞争力。这时，就有了虚拟化技术的再次发展。以 VMware 为代表的虚拟化软件产商率先实施了以虚拟机监视器为中心的软件解决方案，为虚拟化技术在 x86 计算机环境的发展开辟了道路。

最近的十几年间，诸多厂商（如微软、Intel 公司、AMD 公司等）都开始进行虚拟化技术的研究。为了与 VMware 展开直接的竞争，微软开发了 Hyper-V 技术。微软凭借其强大的技术支持，成为 VMware 竞争小企业市场的主要对手。同时，虚拟化技术的飞速发展也引起了芯片厂商的重视，Intel 公司和 AMD 公司在 2006 年以后都逐步在其 x86 处理器中增加了硬件虚拟化功能。

2008 年以后，云计算技术的发展推动了虚拟化技术成为研究热点，由于虚拟化技术能够屏蔽底层的硬件环境，充分利用计算机的软硬件资源，是云计算技术的重要目标之一，虚拟化技术成为切分型云计算技术的核心技术。虚拟化对云计算技术的发展产生重大意义的是基于 x86 架构的服务器虚拟化技术。

3.1.2 虚拟化的描述

现代计算机系统被分为多个自下而上的层次。从下到上依次是裸机（底层硬件）、操作系统，操作系统提供应用程序编程接口及运行在操作系统之上的各种各样的应用程序。虚拟化技术可

以在这些不同层次之间建立虚拟化层，向上提供与真实层次相同或相近的功能，向下只需知道下层的抽象接口，不需要知道下层的具体实现。虚拟化层的引入，必然给系统带来一定的性能损耗，构建、维护虚拟化层也会增加一定的费用。

虚拟化涉及的领域比较多，虚拟化的技术在不断发展，虚拟化的定义也在不断发展中。维基百科中虚拟化的描述为“在计算机技术中，虚拟化（Virtualization）是将计算机物理资源如服务器、网络、内存及存储等予以抽象、转换后呈现出来，使用户可以以比原本的组态更好的方式来应用这些资源。这些资源的新虚拟部分是不受现有资源的架设方式、地域或物理组态所限制的”。

我们可以从虚拟化的对象、过程及其要达到的结果方面来对虚拟化进行如下描述。

（1）对象：计算机的各种资源，这些资源包括基础设施、系统和软件。

（2）过程：将各种资源进行抽象、转换。

（3）结果：为这些资源提供标准的接口来接收输入和提供输出，使用户能以更好的方式来应用这些资源。

3.1.3 虚拟化技术的优势和劣势

1．虚拟化技术的优势

虚拟化技术的出现和发展提高了资源的利用率，使得企业能以更低的成本获得更大的收益。从总体上而言，虚拟化的优势体现在以下方面。

（1）虚拟化技术可以提高资源利用率。

传统的 IT 企业为每一项业务分配一台单独的服务器，服务器的实际处理能力往往远超服务器的平均负载，使得服务器大部分时间都处于空闲状态，造成资源的浪费。而虚拟化技术可以减少必须进行管理的物理资源的数量，隐藏了物理资源的部分复杂性。为了达到资源的最大利用率，虚拟化还把一组硬件资源虚拟化为多组硬件资源，并动态地调整空闲资源，减小服务器的规模。例如，VMware 的用户在使用 VMware 的虚拟基础架构解决方案之后服务器的利用率通常可由原先的 5%～15%提升到 60%～80%。

（2）提供相互隔离、高效的应用执行环境。

虚拟化技术能够实现较简单的共享机制无法实现的隔离和划分，从而对数据和服务进行可控和安全的访问。例如，用户可以在一台计算机上模拟多个不同、相互之间独立的操作系统，这些虚拟的操作系统可以是 Windows 或 Linux 系统。其中的一个或多个子系统遭受攻击而崩溃时，不会对其他系统造成影响。在使用备份机制后，受到攻击的子系统可以快速恢复。

（3）虚拟化可以简化资源和资源的管理。

计算机有硬盘、磁盘等硬件资源和 Web 服务等软件资源。用户对计算机资源进行访问是通过标准接口来进行的。使用标准接口的好处是用户不用知道虚拟资源的具体实现。底层的基础设施发生变化时，只要标准接口没有发生变化，用户基本上感受不到这种变化。这是因为，与用户直接接触的是标准接口，虽然底层的具体实现发生改变，但是用户与虚拟资源进行交互

的方式并没有改变。

传统的IT服务器资源是硬件相对独立的个体，对每一种资源都要进行相应的维护和升级，会耗费大量的人力、物力。而虚拟化系统降低了用户与虚拟资源之间的耦合度，利用这种松耦合的关系，管理者可以在对用户影响最小的基础上对资源进行管理。此外，虚拟化系统还将资源进行整合，在管理上相对而言比较方便，在升级时也只需添加动作，从而提高工作效率。

（4）虚拟化技术实现软件和硬件的分离。

用户在同一个计算机系统上可以运行多个软件系统，不同的软件系统通过虚拟机监视器（VMM, Virtual Machine Monitor）来使用底层的硬件资源，从而实现多个软件系统共享同一个硬件资源，达到软件和硬件的分离。这样，在虚拟化的统一的资源池能够运行更多的软件系统，充分利用已有的硬件资源。

2．虚拟化技术的劣势

任何事物都是有利有弊的，虚拟化技术也不例外。物理计算机上的硬件用的时间久了很可能会损坏，其上的软件也要定时地更新，防止病毒的感染。虚拟化技术由于是针对实际的计算机来进行的，虚拟化技术方案的部署、使用也有一些劣势。

（1）可能会使物理计算机负载过重。

虚拟化技术虽然是在虚拟的环境中运行的，但是其并不是完全虚拟的，依然需要硬件系统的支持。以服务器虚拟化为例，一台物理计算机上可以虚拟化出多台客户机，每台客户机上又可以安装多个应用程序。若这些应用程序全部运行的话，就会占用大量的物理计算机的内存、CPU等硬件系统，从而给物理计算机带来沉重的负担，可能会导致物理计算机负载过重，使各虚拟机上的应用程序运行缓慢，甚至系统崩溃。

（2）升级和维护引起的安全问题。

物理计算机的操作系统及操作系统上的各种应用软件都需要不定时地进行升级更新，以增强其抵抗攻击的能力。每台客户机也都需要进行升级更新。一台物理计算机上安装多台客户机，会导致在客户机上安装补丁速度缓慢。如果，客户机上的软件不能及时更新，则很可能会被病毒攻击，带来安全隐患。

（3）物理计算机的影响。

传统的物理计算机发生不可逆转的损坏时，若不是作为服务器出现，则只有其自身受到影响。当采用虚拟化技术的物理计算机发生宕机时，其所有的虚拟机都会受到影响。在虚拟机上运行的业务也会受到一定程度的影响，甚至是损坏。此外，一台物理计算机的虚拟机往往会有相互通信，在相互通信的过程中，可能会导致安全风险。

3.1.4 虚拟化技术的分类

虚拟化技术从计算体系结构的层次上可分为：指令集架构级虚拟化、硬件级虚拟化、操作系统级虚拟化、编程语言级虚拟化和数据库级虚拟化，其比较如表3.1所示。

表 3.1　5 种虚拟化技术的比较

虚拟化类型	虚拟化出的目标对象	所处位置	实例
指令集架构级虚拟化	指令集	指令集架构级	Bochs、VLIW
硬件抽象层虚拟化	计算机的各种硬件	应用层	VMWare、Virtual PC、Xen、KVM
操作系统层虚拟化	操作系统	本地操作系统内核	Virtual Server、Zone、Virtuozzo
编程语言层上的虚拟化	应用层的部分功能	应用层	JVM、CLR
库函数层的虚拟化	应用级库函数的接口	应用层	Wine

1．指令集架构级虚拟化

指令集架构级虚拟化是通过纯软件方法，模拟出与实际运行的应用程序（或操作系统）所不同的指令集去执行，采用这种方法构造的虚拟机一般称为模拟器（Emulator）。模拟器是将虚拟平台上的指令翻译成本地指令集，然后在实际的硬件上执行。其特点是简单、具有鲁棒性、可跨平台。当前比较典型的模拟器系统有 Bochs、VLIW 等。以 Bochs 为例，Bochs 是用 C++编写的模拟 x86 平台的模拟器。用户可以在任何编译运行 Bochs 的平台上模拟 x86 的各种硬件，并且在 Bochs 的仿真平台上可以安装大多数的操作系统。

2．硬件抽象层虚拟化

硬件抽象层虚拟化是指将虚拟资源映射到物理资源，并在虚拟机的运算中使用实实在在的硬件。即使用软件来虚拟一台标准计算机的硬件配置，如 CPU、内存、硬盘、声卡、显卡、光驱等，成为一台虚拟的裸机。这样做的目的是为客户机操作系统呈现和物理硬件相同或类似的物理抽象层。客户机绝大多数指令在宿主机上直接运行，从而提高了执行效率。但是，给虚拟机分配的硬件资源的同时虚拟软件本身也要占用实际硬件资源的，对性能损耗较大。虽然如此，硬件抽象层虚拟化的优点仍不可忽视。硬件抽象层的虚拟机具有以下优点：

（1）高度的隔离性；

（2）可以支持与宿主机不同的操作系统及应用程序；

（3）易于维护及风险低。

比较有名的硬件抽象层虚拟化解决方案有 VMWare、Virtual PC、Xen、KVM 等。以 Xen 为例，Xen 是剑桥大学开发的一个基于 x86 的开源虚拟机监视器，可以在一台物理机上执行多台虚拟机。它特别适用于服务器整合，具有性能高、占用资源少，节约成本等优点。

3．操作系统层虚拟化

操作系统层虚拟化是指通过划分一个宿主操作系统的特定部分，产生一个个隔离的操作执行环境。操作系统层的虚拟化是操作系统内核直接提供的虚拟化，虚拟出的操作系统之间共享

底层宿主操作系统内核和底层的硬件资源。操作系统虚拟化的关键点在于将操作系统与上层应用隔离开，将对操作系统资源的访问进行虚拟化，使上层应用觉得自己独占操作系统。操作系统虚拟化的好处是实现了虚拟操作系统与物理操作系统的隔离并且有效避免物理操作系统的重复安装。比较有名的操作系统虚拟化解决方案有 Virtual Server、Zone、Virtuozzo 及虚拟专用服务器（Virtual Private Server，VPS）。VPS 是利用虚拟服务器软件在一台物理机上创建多个相互隔离的小服务器。这些小服务器本身就有自己的操作系统，其运行和管理与独立主机完全相同。其可以保证用户独享资源，且可以节约成本。

操作系统虚拟化看似与硬件虚拟化出的虚拟机上安装的操作系统一样，都是产生多个操作系统，但操作系统虚拟化与硬件虚拟化之间还是有很多不同之处，区别如下。

（1）操作系统虚拟化是以原系统为模板，虚拟出的是原系统的副本，而硬件虚拟化虚拟的是硬件环境，然后真实地安装系统。

（2）操作系统虚拟化虚拟出的系统只能是物理操作系统的副本，而硬件虚拟化虚拟出的系统可以为不同的系统，如 Linux、Windows 等。

（3）虚拟出的系统间关系不同，操作系统虚拟化虚拟的多个系统有较强的联系。例如，多个虚拟系统能够同时被配置。原系统发生了改变，所有虚拟出的系统都会改变。而硬件虚拟化虚拟的多个系统是相互独立的，与原系统也没有联系，原系统的损坏不会殃及虚拟系统。

（4）性能损耗不同，操作系统虚拟化虚拟出的系统都是虚拟的，性能损耗低，而硬件虚拟化是在硬件虚拟层上实实在在安装的操作系统，性能损耗高。

4．编程语言层上的虚拟化

计算机若不安装操作系统和其他软件的话，就是一台裸机。操作系统和其他软件相对于裸机而言都是应用程序。编程语言层上的虚拟机是在应用层上创建的，并支持一种新定义的指令集。这一类虚拟机运行的是针对虚拟体系结构的进程级作业，通常这种虚拟机是作为一个进程在物理计算机系统中运行的，使得用户感觉不到应用程序是在虚拟机上运行的。这种层次上的虚拟机主要有 JVM(Java Virtual Machine)和 CLR(Common Language Runtime)。以 JVM 为例，JVM 是通过在物理计算机上仿真模拟计算机的各种功能来实现的，是虚拟出来的计算机。JVM 使 Java 程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上进行无缝迁移。

5．库函数层的虚拟化

在操作系统中，应用程序的编写会使用由应用级的库函数提供的一组 API 函数。这些函数隐藏了一些操作系统的相关底层细节，降低了程序员的编程难度。库函数层的虚拟化就是对操作系统中的应用级库函数的接口进行虚拟化，创造出了不同的虚拟化环境。使得应用程序不需要修改，就可以在不同的操作系统中迁移。当然不同的操作系统库函数的接口不一样。如属于这类虚拟化的 Wine，是利用 API 转换技术做出 Linux 与 Windows 相对应的函数来调用 DLL，从而能在 Linux 系统中运行 Windows 程序。

3.2 常见虚拟化软件

3.2.1 VirtualBox

VirtualBox 是一款开源免费的虚拟机软件，使用简单、性能优越、功能强大且软件本身并不臃肿，VirtualBox 是由德国软件公司 InnoTek 开发的虚拟化软件，现隶属于 Oracle 旗下，并更名为 Oracle VirtualBox。其宿主机的操作系统支持 Linux、Mac、Windows 三大操作平台，在 Oracle VirtualBox 虚拟机里面，可安装的虚拟系统包括各个版本的 Windows 操作系统、Mac OS X（32 位和 64 位都支持）、Linux 内核的操作系统、OpenBSD、Solaris、IBM OS2 甚至 Android 4.0 系统等操作系统，在这些虚拟的系统里面安装任何软件，都不会对原来的系统造成任何影响。它与同类的 VMware Workstation 虚拟化软件相比，VirtualBox 对 Mac 系统的支持要好很多，运行比较流畅，配置比较傻瓜化，对于新手来说也不需要太多的专业知识，很容易掌握，并且免费这一点就足以比商业化的 VMware Workstation 更吸引人，因此 VirtualBox 更适合预算有限的小环境。

3.2.2 VMware Workstation

VMware Workstation 是一款功能强大的商业虚拟化软件，和 VirtualBox 一样，仍然可以在一个宿主机上安装多个操作系统的虚拟机，宿主机的操作系统可以是 Windows 或 Linux，可以在 VMware Workstation 中运行的操作系统有 DOS、Windows 3.1、Windows 95、Windows 98、Windows 2000、Linux、FreeBSD 等。VMware Workstation 虚拟化软件虚拟的各种操作系统仍然是开发、测试 、部署新的应用程序的最佳解决方案。VMware Workstation 占的空间比较大，VMware 公司同时还提供一个免费、精简的 Workstation 环境——VMware Player，可在 VMware 官方网站下载使用。对于企业的 IT 开发人员和系统管理员而言， VMware Workstation 在虚拟网络、实时快照、拖曳共享文件夹、支持 PXE 等方面的特点使它成为必不可少的工具。

总体来看，VMware Workstation 的优点在于其计算机虚拟能力，物理机隔离效果非常优秀，它的功能非常全面，倾向于计算机专业人员使用，其操作界面也很人性化；VMware Workstation 的缺点在于其体积庞大，安装时间耗时较久，并且在运行使用时占用物理机的资源较大。

3.2.3 KVM

KVM（Kernel-based Virtual Machine）是一种针对 Linux 内核的虚拟化基础架构，它支持具有硬件虚拟化扩展的处理器上的原生虚拟化。最初，它支持 x86 处理器，但现在广泛支持各种处理器和操作系统，包括 Linux、BSD、Solaris、Windows、Haiku、ReactOS 和 AR-OS 等。基于内核的虚拟机 （KVM） 是针对包含虚拟化扩展（Intel VT 或 AMD-V）的 x86 硬件上的

Linux 的完全原生的虚拟化解决方案。对半虚拟化（Paravirtualization）的有限支持也可以通过半虚拟网络驱动程序的形式用于 Linux 和 Windows Guest 系统。

尽管 KVM 是一个相对较新的虚拟机管理程序，但这个随主流 Linux 内核发布的轻量型模块提供简单的实现和对 Linux 重要任务的持续支持。KVM 使用很灵活，Guest 操作系统与集成到 Linux 内核中的虚拟机管理程序通信，以直接寻址硬件，无需修改虚拟化的操作系统。这使得 KVM 成为更快的虚拟机解决方案。KVM 的补丁与 Linux 内核兼容，KVM 在 Linux 内核本身内实现，这进而简化对虚拟化进程的控制，但是没有成熟的工具可用于 KVM 服务器的管理，KVM 仍然需要改进虚拟网络的支持、虚拟存储的支持，并且增强安全性、高可用性、容错、电源管理、HPC/实时支持、虚拟 CPU 可伸缩性、跨供应商兼容性、VM 可移植性。

3.3 系统虚拟化

1．什么是系统虚拟化

系统虚拟化是指在一台物理计算机系统上虚拟出一台或多台虚拟计算机系统。虚拟计算机系统（简称虚拟机）是指使用虚拟化技术运行在一个隔离环境中的具有完整硬件功能的逻辑计算机系统，包括操作系统和应用程序。一台虚拟机中可以安装多个不同的操作系统，并且这些操作系统之间相互独立。虚拟机和物理计算机系统可以有不同的指令集架构，这样会使得虚拟机上的每一条指令都要在物理计算机上模拟执行。显而易见，会导致性能低下。所以，我们一般使虚拟机的指令集架构与物理计算机系统相同。这样大部分指令都会在处理器上直接运行，只有那些需要虚拟化的指令才会在虚拟机上运行。

2．系统虚拟化的典型特征

说到虚拟机，就不得不提到虚拟机的特性。1974 年，Popek 和 Goldberg 在发表的文章“Formal Requirements for Virtualizable Third Generation Architectures”中指出虚拟机可以看作是物理机的一种高效隔离的复制，并指出虚拟机有同一性、高效性、受控性的 3 个典型特征。

同一性是指虚拟机的运行环境和物理机的运行环境在本质上应该是相同，表现形式上可以有所差别。

高效性是指软件在虚拟机上运行时，大部分是在硬件上运行的，只有少数是在虚拟机中运行的，从而在虚拟机中运行的软件的性能接近在物理机上运行的性能。

资源受控是指 VMM 完全控制和管理系统资源。

3．系统虚拟化的优点

系统虚拟化提供了多个相互隔离的执行环境，虚拟机之间隔离性，虚拟机与底层硬件之间的无关性所带来的好处是很难估量的。此外，虚拟化层作为特权层能够提供一些特有的功能。

（1）硬件无关性。

虚拟机与底层硬件之间是虚拟化层，其与底层硬件之间并没有直接的联系。所以只要另一台计算机提供相同的虚拟硬件抽象层，一个虚拟机就可以无缝地进行迁移。

（2）隔离性。

使用虚拟机，应用软件可以独立地在虚拟机上运行，不受其他虚拟机的影响。即使其他的虚拟机崩溃，也可以正常运行。这种隔离性的好处是：可以在一台物理机虚拟出的多台虚拟机上进行不同的操作，相互之间没有影响。

（3）多实例。

在一台物理机上可以运行多台虚拟机，而一台虚拟机上又可以安装多个操作系统。不同的虚拟机的繁忙、空闲时间又不同，这样虚拟机交错使用物理计算机的硬件资源，资源利用率比较高。

（4）特权功能。

系统虚拟化的虚拟化层是在本地硬件与虚拟机之间，其将下层的资源抽象成另一种形式的资源，提供给上层的虚拟机使用。虚拟化层拥有更高的特权体现在：虚拟化层中添加的功能不需要了解客户机的具体语义，实现起来更加容易，并且添加的功能具有较高的特权级，不能被客户机绕过。

3.3.1 服务器虚拟化

系统虚拟化的最大价值在于服务器虚拟化。服务器虚拟化是将系统虚拟化技术应用于服务器上，将一台或多台服务器虚拟化为若干台服务器使用。现在，数据中心大部分使用的是 x86服务器。一个数据中心可能有成千上万台 x86 服务器。以前，出于性能、安全等方面的考虑，一台服务器只能执行一个服务，导致服务器利用率低下。服务器虚拟化是在一台物理服务器上虚拟出多个虚拟服务器，每个虚拟服务器执行一项任务。这样的话，服务器的利用率相对较高。

1．服务器虚拟化的分类

服务器虚拟化按虚拟的服务器台数可以分为以下 3 种类型。

（1）将一台服务器虚拟成多台服务器，即将一台物理服务器分割成多个相互独立、互不干扰的虚拟环境。

（2）服务器整合，就是多个独立的物理服务器虚拟为一个逻辑服务器，使多台服务器相互协作，处理同一个业务。

（3）服务器先整合、再切分，就是将多台物理服务器虚拟成一台逻辑服务器，然后再将其划分为多个虚拟环境，即多个业务在多台虚拟服务器上运行。

2．服务器虚拟化所需的技术

物理服务器有其不可缺少的关键部件，如 CPU、I/O 等。那么，这些关键部件是如何进行虚拟化的呢？服务器虚拟化的关键技术是对 CPU、内存、I/O 硬件资源的虚拟化。下面对这 3 种硬件资源的虚拟化进行介绍。

（1）CPU 虚拟化。

CPU 虚拟化技术是把物理 CPU 抽象成虚拟 CPU，任意时刻一个物理 CPU 只能运行一个虚拟 CPU 指令。每个客户操作系统可以使用一个或多个虚拟 CPU。在这些客户操作系统之间，

虚拟 CPU 的运行相互隔离，互不影响。

在纯软件的 CPU 虚拟化中，有全虚拟化和半虚拟化两种不同的软件方案。全虚拟化是采用二进制动态翻译技术（Dynamic Binary Translation）来解决客户操作系统的特权指令问题。半虚拟化是通过修改客户操作系统来解决虚拟机执行特权指令的问题，即将所有敏感指令替换为对底层虚拟化平台的超级调用。这两种方案都会增加系统的复杂性和性能开销。

原来的 x86 CPU 不能有效地支持虚拟化，那时，CPU 虚拟化只能在软件层面上进行。随着硬件技术的发展，硬件的性能有了很大的提高，现在主流的 x86 CPU 开始在硬件层面上支持 CPU 虚拟化，从而就有了 CPU 的硬件辅助虚拟化。CPU 的硬件辅助虚拟化是在 CPU 中加入新的指令集和处理器运行模式来支持 CPU 虚拟化，使得系统软件能更加容易、高效地实现虚拟化功能。

CPU 的硬件辅助虚拟化主要有 Intel VT-x 和 AMD。以 VT-x 为例，VT-x 的原理是：首先，其引入了根（VMX root operation）和非根（VMX non-root operation）两种操作模式，这两种模式统称为 VMX 操作模式。根操作模式是 VMM 运行所处的模式，其行为和早期的没有 VT-x 技术的 x86 CPU 相同。非根操作模式是客户机运行时所处的模式，提供了一个支撑虚拟机运行所需的 CPU 环境。这两种操作模式都有特权级 0~特权级 3，共 4 种特权级。在 VT-x 中，从非根操作模式到根操作模式的转换形式称为 VM-Exit. 而从根操作模式到非根操作模式的转换形式为 VM-Entry。此外，VT-x 还引入了保存虚拟 CPU 相关状态的 VMCS 来更好地支持 CPU 虚拟化。

（2）内存虚拟化。

内存虚拟化是对宿主机的真实物理内存统一管理，虚拟化成虚拟的物理内存，然后分别供若干个虚拟机使用，使得每个虚拟机拥有各自独立的内存空间。

对于真实的操作系统而言，内存是从物理地址 0 开始的，且是连续的，至少在一些大粒度上是连续的。在虚拟化中，所有的客户操作系统可能会同时使用起始地址是 0 的物理内存，为了满足所有的客户操作系统的起始物理地址都是 0 且它们内存地址的连续性，VMM 引入了一层新的地址空间——客户机物理地址空间。

虚拟机监视器（VMM）通过虚拟机内存管理单元（Memory Management Unit，MMU）来管理虚拟机内存，即其负责分配和管理每个虚拟机的物理内存。客户机操作系统看到的是一个虚拟的物理内存地址空间（即客户机物理地址空间），不再是真正的物理内存地址空间。有了客户机物理地址空间就形成了两层地址映射即应用程序所对应的客户机虚拟地址空间到客户机物理地址的映射，客户机物理地址到宿主机物理地址的映射。前一种映射是由客户机操作系统完成的，后一种是由 VMM 通过动态地维护镜像页表来管理的。

（3）I/O 虚拟化。

在一台虚拟机上可以安装多个操作系统，这些客户操作系统都会对外设资源进行访问。但是，外设资源是有限的，为了使所有的客户操作系统都能访问外设资源，虚拟机监视器需通过 I/O 虚拟化的方式复用有限的外设资源。此时，VMM 截获客户操作系统对外设的访问请求，然后通过软件的方式模拟真实外设的效果。但是，并不要求完整地虚拟化出所有外设的所有接口。

I/O 虚拟化的第一步是发现设备，设备的发现取决于被虚拟的设备类型。设备类型不同，设备的发现方式也不同。以模拟一个完全虚拟的设备为例，这种虚拟设备所处的总线类型完全由 VMM 自行决定，VMM 可以自定义一套虚拟总线协议，也可以将虚拟设备挂在 PCI 总线上。第二步是截获访问，虚拟设备已经发现，此时 VMM 的工作是使客户机操作系统对其进行访问。VMM 会根据设备的不同性能提供不同的截获方式。例如，对于直接分配给客户操作系统并有端口 I/O 资源的设备，VMM 的处理方式是把该设备所属的端口 I/O 从 I/O 位图中打开，访问就会被处理器发送给系统总线，最后到达目标物理设备。

在 I/O 设备中有一种比较特殊的设备——网卡。网卡除了和一般的 I/O 设备一样作为虚拟机的共享设备外，还要解决虚拟机与外部网络或者虚拟机相互之间的通信问题。网卡虚拟化技术主要分为两类：虚拟网卡技术和虚拟网桥技术。虚拟网卡是指虚拟机中的网卡，是由模拟器通过软件的方法模拟出来的；虚拟网桥是指利用软件方法实现的网桥其作用是在一台服务器中，使多块共享一块物理网卡的虚拟网卡对外表现为多块独立的网卡。

3.3.2 桌面虚拟化

桌面虚拟化依赖于服务器虚拟化，直观上来说就是将计算机的桌面进行虚拟化，是将计算机的桌面与其使用的终端设备相分离。桌面虚拟化为用户提供部署在云端的远程计算机桌面环境，用户可以使用不同的终端设备通过网络来访问该桌面环境，即在虚拟桌面环境服务器上运行用户所需要的操作系统和应用软件。桌面虚拟化的应用软件安装在云端服务器上，即使本地服务器上没有应用软件，用户依然可以通过虚拟桌面来访问相关的应用。

1．桌面虚拟化的优势

（1）更灵活的访问和使用。

传统的计算机桌面，需要在特定的设备上使用，例如，某用户的计算机桌面上安装了 PhotoShop 软件，若要使用，只能用自己的那台计算机。虚拟桌面不是本人直接安装在设备上，而是部署在远程服务器上的。任何一台满足接入要求的终端设备在任何时间、任何地点都可以进行访问。例如，拥有虚拟桌面的用户，在上班的时候可以使用单位提供的瘦客户端设备来访问虚拟桌面，在出行的路上可以使用智能手机、平板计算机上安装的客户端软件来访问虚拟桌面，更加方便、快捷。

（2）更低的用户终端配置。

虚拟桌面部署在远程服务器上，所有的计算都在远程服务器上进行，而终端设备主要是用来显示远程桌面内容。所以，终端设备没有必要拥有与远程服务器相似的配置，对其配置要求更低、维护相对而言也更加容易。

（3）更便于集中管控终端桌面。

虚拟桌面并不是没有自己的个人桌面，其完全可以与本地的个人桌面同时存在，两者可以互不干扰。使用虚拟桌面，运行商将所有的桌面管理放在后端的数据中心中，数据中心可以对桌面镜像和相关的应用进行管理、维护。而终端用户不用知道具体的管理和维护，就可以使用

经过维护后的桌面。

（4）更高的数据安全性。

用户在虚拟桌面上所做的应用是在后台的数据中心中执行的，所产生的数据也存储在数据中心，并没有存储在用户的终端设备上。从而，用户终端设备的损坏对数据没有影响。此外，由于传统的物理桌面会接入内部网，一旦一个终端感染病毒，就可能殃及整个内部网络。而虚拟桌面的镜像文件受到感染，受影响的只是虚拟机，能很快地得到清除和恢复。

（5）更低的成本。

虚拟桌面简化了用户终端，用户可以选择配置相对较低的终端设备，从而节省购买成本。同时，传统的每台计算机上都要有一个桌面环境。而且这些计算机分布在世界各地。管理起来比较困难，管理成本也比较高。而虚拟桌面及其相关应用的管理和维护都是在远程服务器端运行的，成千上万的用户可以使用同一个虚拟桌面，从而降低了管理和维护的成本。

2．虚拟桌面的解决方案

用户开始使用桌面已经很多年了，最先是在自己的计算机上使用，现在已经形成了基于虚拟桌面基础架构（Virtual Desktop Infrastructure，VDI）和基于服务器计算技术（Server-Based Computing，SBC）两种技术解决方案，这两种技术方案都是一种端到端的桌面管理解决方案，如表 3.2 所示。

表 3.2　VDI 解决方案与 SBC 解决方案的比较

项目	VDI	SBC
服务器性能要求	高，需要能支持服务器虚拟化软件的运行	低，只要能部署操作系统及应用软件
用户支持扩展性	低，与服务器上能同时承载的虚拟机个数有关	高，与服务器上能同时支持的应用软件执行实例有关
方案实施复杂性	高，需要在安装和管理服务器虚拟化软件的前提下提供服务	低，只需要以传统方式安装和部署应用软件就可提供服务
桌面交付兼容性	高，支持 Linux 桌面、Windows 桌面等桌面上的应用	低，只支持 Windows 上的应用
桌面安全隔离性	高，依赖于虚拟机之间的安全隔离性	低，依赖于 Windows 操作系统进程之间的安全隔离性
桌面性能隔离性	高，依赖于虚拟机之间的性能隔离性	低，依赖于 Windows 操作系统进程之间的性能隔离性
终端应用程序兼容性	无，每一个桌面都是一个独立的工作站	有，依赖于操作系统的版本
提供服务的性能	高，在一个刀片上只有一个用户或少数几个用户	低，在一个刀片上的用户数相对较多

（1）基于 VDI 的虚拟桌面解决方案。

基于 VDI 的虚拟桌面解决方案是基于服务器虚拟化的，拥有服务器虚拟化的所有优点。其原理是在远程数据中心的服务器上安装虚拟机并在其中部署用户所需要的操作系统及操作系统上的各种应用，此时虚拟桌面就是虚拟机上的操作系统及其上的各种应用。然后通过桌面显示协议将完整的虚拟桌面交付给终端用户使用。终端用户通过一对一的方式连接和控制运行在远端服务器上的实例。

桌面显示协议是指在远程桌面与终端之间所使用的通信协议，用于键盘等输入设备、显示设备等与桌面信息之间的数据传输。桌面显示协议是桌面虚拟化软件的核心部件。当前主流的显示协议包括 RDP（Remote Desktop Protocol）、PCoIP、SPICE、ICA 等。

基于 VDI 的虚拟桌面解决方案，用户可以"暂停"单个虚拟机，然后将它们从一个服务器迁移到另一个服务器。如果服务器端的 Windows XP 是基于 VMware VDI 基础架构的，数据中心的管理员就可以保有一些很酷的灵活性。例如，管理员可以通过管理控制台中的一个按钮来"移动"用户到另一台服务器上。用户会收到一个弹出框，显示"请稍等片刻"，然后服务器将 Windows XP 桌面 VM 的内存内容转储到磁盘，虚拟机将被置备到另一个物理硬件上。这整个过程大概需要不到 30s，而用户会正好回到他们离开的地方。这项技术的另一个用途是，管理员可能有一个额外的"超时"设置。例如，20min 后没有活动的用户会话将被中断（它仍然是在服务器上运行，但是从客户端断开连接）。如果在 1h 后用户仍然没有进行连接，该系统就可以"暂停"会话并转储内存内容到磁盘，然后释放出硬件资源供其他的用户使用。每当该用户连接时，会话将重新开始，无论过了多久，用户都将回到他们离开的地方。

（2）基于 SBC 的虚拟桌面解决方案。

基于 SBC 的虚拟桌面解决方案的原理是在数据中心内的物理机上直接安装、运行操作系统和应用软件，此时的桌面就是服务器上的物理桌面。用户通过和服务器建立的会话对服务器桌面及相关应用进行访问和操作。这类解决方案在服务器上部署的操作系统是必须支持多用户多会话的，并且允许多个用户共享操作系统桌面。同时，用户会话产生的输入/输出数据被封装为桌面交付协议格式在服务器和客户端之间传输。

基于 SBC 的虚拟桌面解决方案，管理员可以在一个数据中心的服务器上运行 50～75 个的桌面会话，并且该服务器的一个实例是由 Windows 来管理的，当使用 VDI 方案时，若有 50～75 个用户就要有 50～75 个操作系统，每一个操作系统都要进行配置、管理、维护，很浪费人力、物力。

3.3.3 网络虚拟化

网络虚拟化并不是一个新的概念，已经提出了十多年了，但是其依然处在早期的运用阶段。由于在不同的虚拟机之间可以建立一个私有的虚拟网络，可以说，网络虚拟化是服务器虚拟化产品的一部分。

网络虚拟化一般是指虚拟专用网。虚拟专用网对网络连接进行了抽象，远程用户可以像物

理连接在组织内部网络的用户一样来访问该网络。虚拟专用网络是通过一个公用网络建立一个临时的、安全的连接，是一条穿过混乱的公用网络的安全、稳定隧道。使用这条隧道可以对数据进行几倍加密达到安全使用互联网的目的。虚拟专用网可以保护网络环境，使用户能够快捷、安全地访问组织内部的网络。

网络虚拟化还有另外一种形式——虚拟局域网。虚拟局域网能把一物理局域网中的节点在逻辑上划分为多个虚拟局域网，或者是把多个物理局域网中的节点划分到一个虚拟局域网中。每一个虚拟局域网都有一组相同需求的计算机工作站，其的工作方式与物理局域网类似。虚拟局域网增强了网络安全和网络管理。例如，在同一个虚拟局域网中的计算机工作站之间的通信与直接在独立的交换机上运行是一样的，虚拟局域网中的广播只有虚拟局域网中的计算机工作站才能收到，控制了不必要的广播风暴的产生。

3.4 使用 KVM 构建虚拟机群

1. 安装KVM

查看 CPU 是否支持虚拟化，这里我们的 CPU 为 4 核，支持虚拟化本机的。

```
grep -E -o 'vmx|svm' /proc/cpuinfo
```

知识：在 Linux 系统中 CPU 的全部信息存储在/proc/cpuinfo 中，使用指令“cat”可以查看其全部信息。如果有“vmx”字样，则表示 CPU 支持 Intel-V 虚拟化技术；如果有“svm”字样，则表示 CPU 支持 AMD-V 虚拟化技术；如果没有任何显示则表示不支持虚拟化。

安装 KVM。

```
yum install kvm kmod-kvm qemu kvm-qemu-img virt-viewer virt-manager libvirt
libvirt-python python-virtinst
```

查看 KVM 模块。

```
lsmod | grep kvm
```

如果显示如下信息，则表示 KVM 安装完毕。

```
kvm_intel           53484  0
kvm                316506  1 kvm_intel
```

启动 libvirtd 服务，并在下次启动时自动启动。

```
service libvirtd start
chkconfig libvirtd on
```

依次单击“应用程序”→“系统工具”→“虚拟机管理器”，启动虚拟机管理器。

2. 新建虚拟机

在 KVM 中新建虚拟机可以采用从本地安装、网络安装、网络引导和导入已有系统镜像等方式进行，如图 3.1 所示。这里我们安装一台虚拟机，并复制生成虚拟机，形成由 3 个虚拟节点构成的集群。

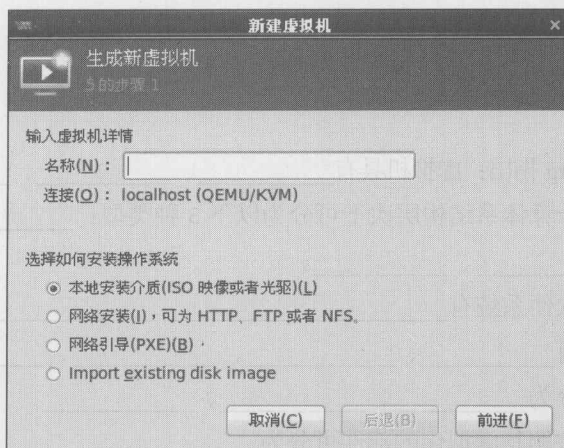


图 3.1 新建虚拟机

3. 管理虚拟机

在 KVM 虚拟机管理器中同时启动 3 个虚拟机，如图 3.2 所示。

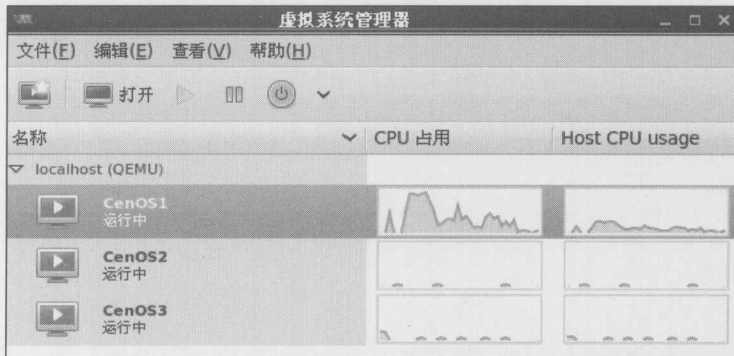


图 3.2 在 KVM 管理器中同时启动 3 个虚拟机

选中其中的一个虚拟机，我们可以进入虚拟机管理设置界面，如图 3.3 所示。

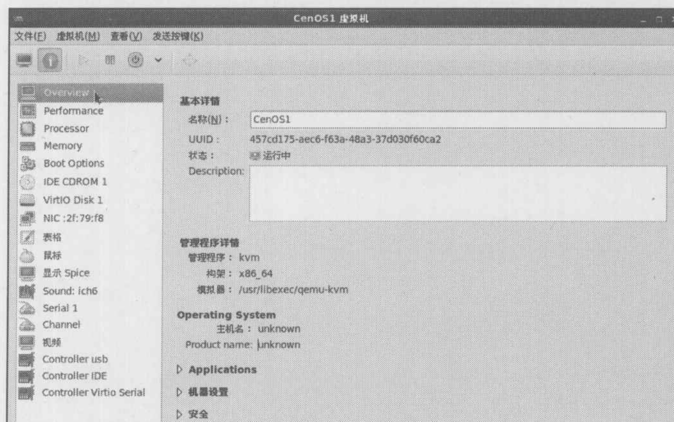


图 3.3 KVM 虚拟机管理设置

练习题

1. Popek 和 Goldberg 指出：虚拟机具有__________、__________、__________3 个特点。
2. 虚拟化技术从计算体系结构层次上可分为以下 5 种类型：__________、__________、__________、__________、__________。
3. 常用的虚拟化软件系统有__________、__________、__________。
4. 系统虚拟化具有__________、__________、__________、__________等优点。
5. 系统虚拟化可分为__________、__________、__________。
6. 服务器虚拟化按照其虚拟化的部件可分为__________、__________、__________。
7. 什么是广义虚拟化技术？
8. 虚拟化技术有哪些优势和劣势？

PART 4

第4章 集群系统基础

当前云计算技术领域存在两个主要技术路线，一个是基于集群技术的云计算资源整合技术，另一个是基于虚拟机技术的云计算资源切分技术。基于集群技术的云计算资源整合技术路线将分散的计算和存储资源整合输出，主要依托的技术为分布式计算技术。集群技术从传统的高性能计算逐步走向云计算和大数据领域，集群架构是当前高性能计算的主流架构，然而无独有偶，集群架构也是大数据领域技术的主流架构，大数据可以认为是面向计算的高性能计算技术，集群技术是大数据系统的重要技术。Google、Hadoop、Storm、HPCC 等系统都采用了集群技术，其资源整合是跨物理节点的。学习集群技术的基本知识对理解云计算大数据技术有很好的作用，只有这样在学习时才能知其所以然。

4.1 集群系统的基本概念

并行计算发展到现在的集群架构成为了主流，首先提出云计算概念的 Google 公司其系统的总体结构就是基于集群的，Google 公司的搜索引擎同样就是利用上百万的服务器集群构成的，这些服务器通过软件结合在一起，共同为遍布于全世界的用户提供服务。从云计算的角度看，Google 公司的系统整合了上百万的服务器计算和存储资源通过网络通道将自己的搜索服务提供给用户。利用集群构建云计算系统为云计算资源池的整合提供了最大的想象力，资源池的大小没有任何原则上的限制。

集群系统是一组独立的计算机（节点）的集合体，节点间通过高性能的互联网络连接，各节点除了作为一个单一的计算资源供交互式用户使用外，还可以协同工作，并表示为一个单一的、集中地计算资源，供并行计算任务使用。集群系统是一种造价低廉、易于构建并且具有较好可扩放性的体系结构。

近年来，集群系统之所以发展如此迅速，主要是因为：

① 作为集群节点的工作站系统的处理性能越来越强大，更快的处理器和更高效的多 CPU 机器将大量进入市场；

② 随着局域网上新的网络技术和新的通信协议的引入，集群节点间的通信能获得更高的带宽和较小的延迟；

③ 集群系统比传统的并行计算机更易于融合到已有的网络系统中去；

④ 集群系统上的开发工具更成熟，而传统的并行计算机上缺乏一个统一的标准；

⑤ 集群系统价格便宜并且易于构建；

⑥ 集群系统的可扩放性良好，节点的性能也很容易通过增加内存或改善处理器性能获得提高。

集群系统具有以下重要特征：

① 集群系统的各节点都是一个完整的系统，节点可以是工作站，也可以是 PC 或 SMP 器；

② 互联网络通常使用商品化网络，如以太网、FDDI、光纤通道和 ATM 开关等，部分商用集群系统也采用专用网络互联；

③ 网络接口与节点的 I/O 总线松耦合相连；

④ 各节点有一个本地磁盘；

⑤ 各节点有自己的完整的操作系统。

集群系统作为一种可扩放并行计算体系，与 SMP、MPP 体系具有一定的重叠性，三者之间的界限是比较模糊的，有些 MPP 系统如 IBM SP2，采用了集群技术，因此也可以把它划归为集群系统。在表 4.1 中给出了这三种体系特性的比较，其中 DSM 表示分布式共享内存。

MPP 通常是一种无共享(Shared-Nothing)的体系结构，节点可以有多种硬件构成方式，不过大多数只有主存和处理器。SMP 可以认为是一种完全共享(Shared-Everything)的体系结构，所有的处理器共享所有可用的全局资源(总线、内存和 I/O 等)。对于集群来说，集群的节点复杂度通常比 MPP 高，因为各集群节点都有自己的本地磁盘和完整的操作系统；MPP 的节点通常没有磁盘，并且可以只是使用一个微内核，而不是一个完整的操作系统；SMP 服务器则比一个集群节点要复杂，因为它有更多的外设终端，如终端、打印机和外部 RAID 等。

表 4.1 SMP、MPP、集群的比较一览表

系统特征	SMP	MPP	集群
节点数量(N)	$\leqslant O(10)$	$O(100) \sim O(1000)$	$\leqslant O(10)$
节点复杂度	中粒度或细粒度	细粒度或中粒度	中粒度或粗粒度
节点间通信	共享存储器	消息传递或共享变量(有 DSM 时)	消息传递
节点操作系统	1	N(微内核)和 1 个主机 OS(单一)	N(希望为同构)
支持单一系统映像	永远	部分	希望
地址空间	单一	多或单一(有 DSM 时)	多个
作业调度	单一运行队列	主机上单一运行队列	协作多队列
网络协议	非标准	非标准	标准或非标准
可用性	通常较低	低到中	高可用或容错
性能/价格比	一般	一般	高
互联网络	总线/交叉开关	定制	商用

集群系统不仅带来了上述研究方向，同时也带来了不少具有挑战性的设计问题，如可用性、好用性、良好的性能、可扩放性等。所以在集群系统的设计中要考虑 5 个关键问题：可用性、单一系统映像、作业管理、并行文件系统和高效通信。

（1）可用性：如何充分利用集群系统中的冗余资源，使系统在尽可能长的时间内为用户服务。集群系统有一个可用性的中间层，它使集群系统可以提供检查点、故障接管、错误恢复以及所有节点上的容错支持等服务。

（2）单一系统映像 SSI（Single System Image）：集群系统与一组互联工作站的区别在于，集群系统可以表示为一个单一系统。集群系统中也有一个单一系统映像的中间层，它通过组合各节点上的操作系统提供对系统资源的统一访问。

（3）作业管理(Job Management)：因为集群系统需要获得较高的系统使用率，集群系统上的作业管理软件需要提供批处理、负载平衡、并行处理等功能。

（4）并行文件系统 PFS：由于集群系统上的许多并行应用要处理大量数据，需进行大量的 I/O 操作，而这些应用要获得高性能，就必须要有一个高性能的并行支持文件。

（5）高效通信(Efficient Communication)：集群系统比 MPP 机器更需要一个高效的通信子系统，因为集群系统有以下几点。①节点复杂度高，耦合不可能像 MPP 那样紧密。②节点间的连接线路比较长，带来了较大的通信延迟，同时也带来了可靠性、时钟扭斜(Clock Shew)和串道(Cross−Talking)等问题。③集群系统一般使用标准通信协议下的商品化网络，标准的通信协议开销比较大，影响到系统的性能，而性能较好的低级通信协议缺乏一个统一的标准。

4.2 集群系统的分类

传统的集群系统可以分为以下 4 类。

（1）高可用性集群系统。

高可用性集群系统通常通过备份节点的使用来实现整个集群系统的高可用性，活动节点失效后备份节点自动接替失效节点的工作。高可用性集群系统就是通过节点冗余来实现的，一般这类集群系统主要用于支撑关键性业务的需要，从而保证相关业务的不间断服务。

（2）负载均衡集群系统。

负载均衡集群系统中所有节点都参与工作，系统通过管理节点（利用轮询算法、最小负载优先算法等调度算法）或利用类似一致性哈希等负载均衡算法实现整个集群系统内负载的均衡分配。

（3）高性能集群系统。

高性能集群系统主要是追求整个集群系统计算能力的强大，其目的是完成复杂的计算任务，在科学计算中常用的集群系统就是高性能集群系统，目前物理、生物、化学等领域有大量的高性能集群系统提供服务。

（4）虚拟化集群系统。

在虚拟化技术得到广泛使用后，人们为了实现服务器资源的充分利用和切分，将一台服务

器利用虚拟化技术分割为多台独立的虚拟机使用，并通过管理软件实现虚拟资源的分配和管理。这类集群系统称为虚拟集群系统，其计算资源和存储资源通常是在一台物理机上。利用虚拟化集群系统可以实现虚拟桌面技术等云计算的典型应用。

目前基于集群系统结构的云计算系统往往是几类集群系统的综合，集群系统式云计算系统既需要满足高可用性的要求又尽可能地在节点间实现负载均衡，同时也需要满足大量数据的处理任务，所以像 Hadoop、HPCC 这类云计算大数据系统中前三类集群系统的机制都存在。而在基于虚拟化技术的云计算系统中采用的往往是虚拟化集群系统。

4.3 单一系统映射

云计算系统需要将计算资源和存储资源形成一个统一的资源池，目前基于集群系统技术的云计算系统如 Hadoop 等，从使用者来看并不需要了解集群系统的具体结构，也不需要分别对集群系统中的每一个节点进行操作，系统对外都有一个统一的接口，这就是单一系统映像技术。单一系统映像 SSI 是集群系统的一个重要特征，使用它使得集群系统在使用、控制、管理和维护上更像一个工作站。单一系统映像可以带来以下好处：

（1） 终端用户不需要了解应用在哪些节点上运行；

（2）操作员不需要了解资源所在地位置；

（3）降低了操作员错误带来的风险，使系统对终端用户表现出更高的可靠性和可用性；

（4）可以灵活地采用集中式或分布式的管理或控制，避免了对系统管理员的高需求；

（5） 大大地简化了系统的管理，一条命令就可以对分布在系统中的多个资源进行操作；

（6）提供了位置独立的消息通信。

单一系统映像包括以下含义。

（1）单一系统。

尽管系统中有多个处理器，用户依然把整个集群系统视为一个单一的系统来使用，例如，与分布式系统不一样，用户可以告诉系统：“用 4 个处理器来执行我的应用程序”。

（2）单一控制。

逻辑上，最终用户或系统用户使用的服务都来自只有惟一接口的同一个地方，例如，一个用户将批处理作业提交到一个队列集，系统管理员就可以从一个单一的控制点配置集群系统的所有软、硬件组件。

（3）对称性。

用户可以从任一个节点上获得集群服务，也就是说，对于所有的节点和所有的用户，除了那些对一般访问权限作保护的服务和功能外，所有的集群服务和功能性都是对称的。

（4）位置透明。

用户不用了解真正执行服务的物理设备位置。

在云计算系统中实现单一系统映射可以保证云计算复杂的集群结构及集群节点间的相互关系对于用户是透明的，系统向用户屏蔽了内部的复杂性，云计算系统的使用者只需要面对一

个统一的访问接口就能实现对云计算系统资源的访问。这种做法使云计算系统从逻辑上看上去更像一个统一的巨大的资源池，资源池大小的动态变化及结构被系统所封装。

4.4 Beowulf 集群

Beowulf 集群是一种用作并行计算的集群架构，通常是由一台主节点和一台以上的子节点通过以太网或其他网络连接的系统，它采用市面上可以购买的普通硬件（如装有 Linux 的 PC）、标准以太网卡和交换机，它不包含任何特殊的硬件设备，可以重新组建。Beowulf 一词起源于一首现存的最古老的英语史诗，比喻以较低的成本实现与千百万用户之间的计算机资源共享。Beowulf 集群的出现为并行计算技术的普及提供了可能，使从前只有高端用户才有机会使用的高性能计算系统现在可以在普通实验室使用。

1994 年第一个 Beowulf 系统诞生时使用的是 100MHz 的 Intel 80486 芯片 DX4，在 1994 年 Beowulf 系统的主要瓶颈在于处理器的速度。1998 年在美国 Los Alamos 国家实验室的一个 Beowulf 集群 Avalon 用 140 块 533 MHz Alpha 微处理器（21164A）建成。每个节点有 256 MB 内存和 3 GB 的硬盘空间。这些节点通过快速以太网的 PCI 卡连接。它运行在 RedHat Linux 5.0 上。依据 Linpack benchmark 测试结果，Avalon 集群的运行速度为每秒 477 亿次浮点运算（47.7 Gflops ）。它价值大约 31.3 万美元，并且在价格性能比方面，它排在 Silicon Graphics 的 Origin 2000 64 位处理器之上，后者有相同数量的 Gflops 而价值大约 180 万美元。但随着处理器远大于存储器的速度及总线带宽的速度发展，如今带宽已经成为制约 Beowulf 系统性能的主要因素，系统的综合效率对系统内的带宽有很强的依赖性。

Beowulf 系统这种诞生于面向计算时代的产物，在云计算时代再次被人们所重视，Google 在早期单次点击所获得收益很小的情况下不得已采用了廉价服务器来构建自己的搜索系统，却意外地成就了云计算的出现。开源的 Hadoop 系统也继承了这一理念，利用廉价服务器搭建大数据处理系统。容错机制的出现使系统对服务器可靠性的要求降低，单个节点的性能和失效对整个系统的性能和正常工作影响很小。

Beowulf 系统的特点如下。

（1）Beowulf 系统通常由一个管理节点和多个计算节点构成。它们通过以太网(或其他网络)连接。管理节点监控计算节点，通常也是计算节点的网关和控制终端。当然它通常也是集群系统文件服务器。在大型的集群系统中，由于特殊的需求，这些管理节点的功能也可能由多个节点分摊。

（2）Beowulf 系统通常由最常见的硬件设备组成，例如，PC、以太网卡和以太网交换机。Beowulf 系统很少包含用户定制的特殊设备。

（3）Beowulf 系统通常采用那些廉价且广为传播的软件，例如，Linux 操作系统、并行虚拟机（PVM）和消息传递接口（MPI）。

对于 Beowulf 系统我们可以用图 4.1 来形象地表示它与其他并行机系统的区别。

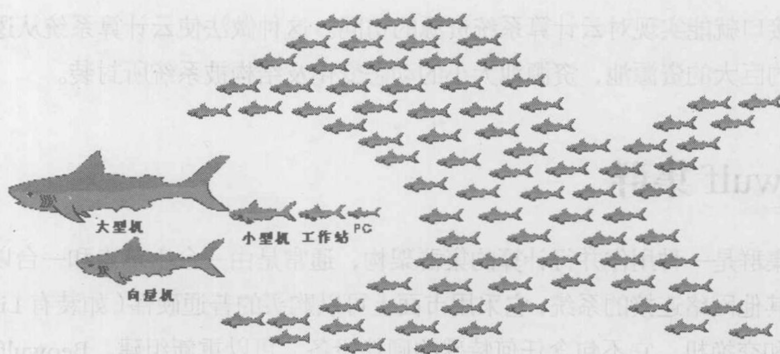


图 4.1 Beowulf 系统与其他并行计算机的对比

图 4.1 形象地表明了 Beowulf 采用廉价通用设备实现了大型计算机的计算能力，体现了群体性能集成的力量，这也是我们采用 Beowulf 系统的实质所在，云计算计算资源池的形成思想与 Beowulf 系统十分相近。

由于 Beowulf 系统采用普通廉价设备构建并行计算机系统，可以在普通实验室环境下实现高性能计算。如图 4.2 所示是成都信息工程学院并行计算实验室采用多台普通 PC 构建的一个 Beowulf 系统。



图 4.2 Beowulf 系统实例

Beowulf 系统的理念在云计算和大数据领域得到了很好的应用，从 Google 的搜索系统，到开源的 Hadoop 都强调自己的系统是面向廉价服务器集群而设计的。单节点的高速度已不是惟一的追求了，利用良好的集群系统架构实现对廉价服务器资源的整合，大大节省了建设的成本。个体的弱小和不稳定与整体的强大和稳定在这里形成了完整的统一。

4.5 集群文件系统

4.5.1 集群文件系统概念

数据的存储一直是人类在不懈研究的内容之一，最早的原始人类采用结绳记事的方式实现数据的记录和存储，后来中国商代利用甲骨作为信息存储的载体（见图 4.3），竹简作为的信息载体的时代大约出现在西周和春秋时期，竹简是中国历史上使用时间最长的信息记录载体之一，公元二世纪初，东汉蔡伦改进造纸术成功，纸张从此在长达一千多年的时间里成为了主要的信息记录载体，直到今天我们仍然在使用纸张这一信息记录载体。

计算机的出现使信息的记录方式再次发生了巨大的变化，计算机的信息记录方式从穿孔纸带、磁带、磁鼓到硬盘、光盘、Flash 芯片，几十年的时间使人类对信息的记录能力实现了多个数量级的跃迁。



图 4.3 信息存储载体——甲骨

信息记录方式可以说一直伴随着人类历史的发展，云计算技术的发展文件系统技术是其中的重要部分，数据的存储方式对云计算系统架构有着重要的影响。传统的存储方式一般是基于集中部署的磁盘阵列，这种存储方式结构简单使用方便，但数据的集中存放在数据使用时不可避免地会出现数据在网络上的传输，这给网络带来了很大的压力。随着大数据技术的出现，面向数据的计算成为云计算系统需要解决的问题之一，集中的存储模式更是面临巨大的挑战，计算向数据迁移这种新的理念，使集中存储风光不在，集群文件系统在这种条件下应运而生。目前常用的 HDFS、GFS、Lustre 等文件系统都属于集群文件系统。

集群文件系统存储数据时并不是将数据放置于某一个节点存储设备上，而是将数据按一定的策略分布式地放置于不同物理节点的存储设备上。集群文件系统将系统中每个节点上的存储空间进行虚拟的整合，形成一个虚拟的全局逻辑目录，集群文件系统在进行文件存取时依据逻辑目录按文件系统内在的存储策略与物理存储位置对应，从而实现文件的定位。集群文件系统相比传统的文件系统要复杂，它需要解决在不同节点上的数据一致性问题及分布式锁机制等问题，所以集群文件系统一直是云计算技术研究的核心内容之一。

在云计算系统在采用集群文件系统有以下几个优点。

（1）由于集群文件系统自身维护着逻辑目录和物理存储位置的对应关系，集群文件系统是很多云计算系统实现计算向数据迁移的基础。利用集群文件系统可以将计算任务在数据的存储节点位置发起，从而避免了数据在网络上传输所造成的拥塞。

（2）集群文件系统可以充分利用各节点的物理存储空间，通过文件系统形成一个大规模的存储池，为用户提供一个统一的可弹性扩充的存储空间。

（3）利用集群文件系统的备份策略、数据切块策略可以实现数据存储的高可靠性以及数据读取的并行化，提高数据的安全性和数据的访问效率。

（4）利用集群文件系统可以实现利用廉价服务器构建大规模高可靠性存储的目标，通过备份机制保证数据的高可靠性和系统的高可用性。

4.5.2 典型的集群文件系统 Lustre

Lustre 是一个应用广泛的集群文件系统，Lustre 系统适合作为并发要求不是很高的云平台的存储模块，因为所有的数据请求都会经过 Lustre 系统元数据服务器。元数据服务器里存放着文件系统的整个基本信息，负责管理整个系统的命名空间，并维护整个文件系统的目录结构、文件名、文件的用户权限，还负责维护文件系统数据的一致性。这就有可能造成整个数据传输在高并发时的瓶颈。因此，Lustre 存储系统最适合的应用场景为：数据请求传输并发不是太高但数据量很大的云平台，例如 HP 公司的“StorageWorks Scalable File Share”（HP SFS，可扩展文件共享），是首款采用 Lustre 技术的商业化产品，而英特尔公司在 2013 年也表示，在发布的 Hadoop 发行版 2.5 版本中加入对 Lustre 的支持能力。由此可见，Lustre 存储系统具备支撑大数据存储的能力，下面先对 Lustre 存储系统本身作一个简单的介绍。

Lustre 存储系统是高性能分布式存储领域中最为著名的系统之一，在全球，有过半的超级计算中心使用 Lustre 存储系统，随着 Lustre 存储系统的发展，越来越多的中大型计算中心和集成平台都在采用 Lustre 存储系统。追朔其起源，Lustre 名字是由 Linux 和 Clusters 派生而来，是为解决海量存储问题而设计的全新文件系统，是 HP、Intel、Cluster File System 公司联合美国能源部开发的 Linux 集群并行文件系统，它来源于卡耐基梅隆大学的 NASD 项目研究工作。Lustre 是基于对象的存储系统，能支持 10000 个节点，PB 级别的存储量，峰值达到 100GB/s 的传输速度，是一个优秀的安全可靠、易于管理的大数量级高性能存储系统。

1. Lustre 存储系统的优点

通过接口提供高性能传输的数据共享，具备并行访问的能力，在数据高并发交互时，系统能对上传、下载的数据操作进行负载均衡，Lustre 存储系统采用双网分离的方式和分布式的锁管理机制来实现并发控制，元数据所走的网络和文件数据传输的所走的网络不同，元数据的 VFS（Virtual File System，存储数据的逻辑视图）部分通常是元数据服务器 10%的负载，剩下 90%的工作由分布在各个节点的数据所在服务器完成。

Lustre 存储系统能够在不影响现行网络的情况下弹性扩充系统的存储容量，还能增加节点

数来扩充网络带宽，具备灵活的扩展性。

在发生访问故障时，Lustre 还能对元数据进行切换，具备访问的稳定可靠性，Lustre 系统中可以拥有两个元数据服务器，采用 Active-Standby（主备方式，指一台服务器处于某种业务的激活状态时，即 Active 状态时，另一台服务器处于该业务的备用状态，即 Standby 状态）的机制，当一台服务器坏掉时，马上切换到另外一个备份服务器。

提供相对健全的接口给用户进行二次开发，以满足不同需求的项目，例如一个在线网盘存储的项目就可以通过 Lustre 存储系统提供的接口进行上层开发，网盘系统里所有用户所存储的文件都最终存储在 Lustre 存储系统中，而对于用户来说则是透明的，用户只需面对网盘系统的视图，同时 Lustre 存储系统本身属于开源项目，对有条件的技术工程师来说，可以进行改造并加以优化。

2．Lustre 存储系统的缺点

与其他很多能支持多操作系统平台安装部署的开源系统不同，Lustre 存储系统只能部署在 Linux 操作系统上，其核心程序很依赖 Linux 操作系统的内核和底层文件操作方法。

对于存储系统来说，节点之间的故障是一个常见的问题，存储系统要考虑节点间故障时如何快速恢复正常。Lustre 在遇到这样的故障时，恢复正常势必要进行切换，切换的条件则需依赖于第三方的心跳技术（heartbeat），heartbeat 包含心跳监测和资源接管两个核心模块，心跳监测可以通过网络链路和串口进行，而且支持冗余链路，它们之间相互发送报文来告诉对方自己当前的状态，如果在指定的时间内未收到对方发送的报文，那么就认为对方失效，这时需启动资源接管模块来接管运行在对方主机上的资源或者服务。

最重要的一点，Lustre 存储系统本身不具备数据自备份的能力，也就意味着使用 Lustre 存储系统作为存储方案时，在系统开发之前，一定要规划一套在其上层开发的存储备份方案，也就是在写入 Lustre 存储资源池的同时，要通过上层软件再把数据写一份到其他存储资源池来备用。

3．Lustre 存储系统的组成

（1）元数据服务器。

元数据服务器(Metadata Server，MDS)负责管理文件系统的基本信息，负责管理整个系统的命名空间，维护整个文件系统的目录结构、用户权限，并负责维护文件系统数据的一致性。通过 MDS 的文件和目录访问管理，Lustre 能够控制客户端对文件系统中文件和目录的创建、删除、修改。Client 可以通过 MDS 读取保存到元数据存储节点 (MDT，Metadata Target，MDT)上的元数据。当 Client 读写文件时，从 MDS 得到文件信息，从 OSS 中得到数据。Client 通过 LNET 协议和 MDS/OSS 通信。在 Lustre 中，MDS 可以有两个，采用 Active-Standby 的容错机制，当其中一个 MDS 不能正常工作时，另外一个后备 MDS 可以启动服务。

（2）元数据存储节点。

元数据存储节点(Metadata Target，MDT)负责存储元数据服务器所要管理的元数据的文件名、目录、权限和文件布局，一个文件系统有且只能有一个 MDT，不同的 MDS 之间共享同一个 MDT。

（3）对象存储服务器。

对象存储服务器(Object Storage Servers，OSS)提供针对一个或多个的本地 OST 网络请求和文件 I/O 服务，OST、MDT 和 Client 可以同时运行在一个节点。但是典型的配置是一个 MDT 专用一个节点，每个 OSS 节点可以有两个或多个 OST，一个客户端可以有大量的计算节点。

（4）对象存储节点。

OST(Object Storage Target，OST)负责实际数据的存储，处理所有客户端和物理存储之间的交互。这种存储是基于对象的，OST 将所有的对象数据放到物理存储设备上，并完成对每个对象的管理。OST 和实际的物理存储设备之间通过设备驱动方式来实现交互。通过驱动程序的事件响应，Lustre 能继承新的物理存储技术及文件系统，实现对物理存储设备的扩展。为了满足高性能计算系统的需要，Lustre 针对大文件的读写进行优化，为集群系统提供较高的 I/O 吞吐率。Lustre 将数据条块化，再把数据分配到各个存储服务器上，以提供比传统 SAN 的“块共享”更为灵活和可靠的共享方式。

（5）客户端。

客户端（Client）通过标准的 POSIX 接口向用户提供对文件系统的访问。对客户端而言，客户端同 OST 进行文件数据的交互，包括文件数据的读写、对象属性的改动等；同 MDS 进行元数据的交互，包括目录管理、命名空间管理等。存储设备是基于对象的智能存储设备，与基于块的 IDE 存储设备不同。客户端在需要访问文件系统的文件数据时，先访问 MDS，获取文件相关的元数据信息，然后就直接和相关的 OST 通信，获取文件的实际数据。客户端通过网络读取服务器上的数据，存储服务器负责实际文件系统的读写操作以及存储设备的连接，元数据服务器负责文件系统目录结构、文件权限和文件的扩展属性以及维护整个文件系统的数据一致性和响应客户端的请求。

4.6 分布式系统中计算和数据的协作机制

计算和存储也是云计算系统研究的核心问题，分布式系统中计算和数据的协作关系非常重要，在分布式系统中实施计算都存在计算如何获得数据的问题，在面向计算时代这一问题并不突出，在面向数据时代计算和数据的协作机制问题就成为了必需考虑的问题，通常这种机制的实现与系统的架构有紧密的关系，系统的基础架构决定了系统计算和数据的基本协作模式，下面以几种常见的分布式系统为例对其计算和数据的协作机制进行分析对比。

4.6.1 基于计算切分的分布式计算

在硬件为核心的时代，高性能计算从以 Cray C-90 为代表的并行向量处理机发展到以 IBM R50 为代表的对称多处理器机(SMP)最终到工作站集群 (COW)及 Beowulf 集群结构，这一过程对应的正是 CPU 等硬件技术的高速发展，可以采用便宜的工作站甚至通用的 PC 来架构高性能系统，完成面向计算的高性能计算任务。

基于消息传递机制的并行计算技术 MPI (Message-Passing Interface)帮助工作站集群和Beowulf 集群实现强大的计算能力，提供了灵活的编程机制。MPI 的标准化开始于 1992 年 4 月，美国并行计算研究中心在弗吉尼亚的 Williamburg 召开消息传递标准的讨论会，讨论了消息传递接口的一些重要基本特征，组建了一个制定消息传递接口标准的工作组，在 1993 年 2 月完成了修订版，这就是 MPI 1.0。1997 年，MPI 论坛发布了一个修订标准，叫作 MPI-2，同时原来的 MPI 更名为 MPI-1。MPICH 为 MPI 标准的一个开源实现，目前已被广泛应用于高性能计算领域。

MPI 将大量的节点通过消息传递机制连接起来，从而使节点的计算能力聚集成为强大的高性能计算，主要面向计算密集的任务。MPI 提供 API 接口，通过 MPI_Send()和 MPI_Recv()等消息通信函数实现计算过程中数据的交换。高性能计算是一种较为典型的面向计算的系统，通常处理的是计算密集型任务，因此在基于 MPI 的分布式系统中并没有与之匹配的文件系统支持，计算在发起前通过 NFS 等网络文件系统从集中的存储系统中读出数据并用于计算。基于 MPI 的分布式系统的典型系统结构如图 4.4 所示。

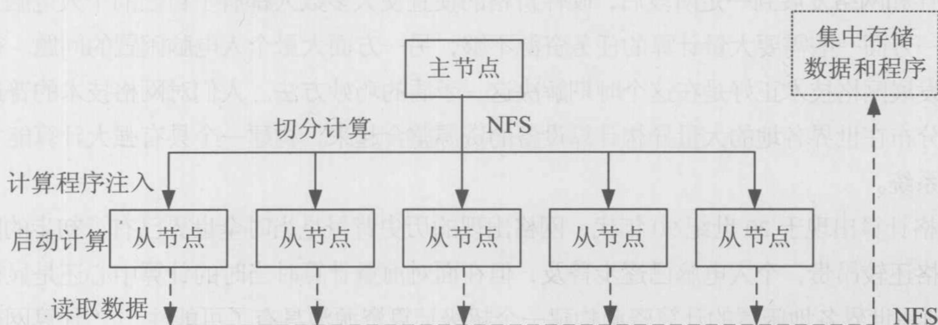


图 4.4 MPI 的典型系统架构

如图 4.4 所示，典型的利用 MPI 实现的分布式计算系统在发起计算时实际上是首先将计算程序由主节点通过 NFS 等网络共享文件系统分发到各子节点内存启动计算，由于没有分布式文件系统的支持，MPI 一般不能直接从节点存储设备上读取数据，计算程序在子节点发起后只有通过网络共享文件读取需要处理的数据来进行计算，在这里数据和计算程序一般都是被集中存储在阵列等专门的存储系统中。这一过程并没有计算寻找数据的过程，计算程序只是按设计要求先被分发给了所有参与计算的节点。在进行 MPI 并行程序设计时，程序设计者需要事先将计算任务本身在程序中进行划分，计算程序被分配到节点后根据判断条件启动相应的计算工作，计算中需要进行节点间的数据交换时通过 MPI 提供的消息传递机制进行数据交换。由于 CPU 的运行速度远远大于网络数据传输的速度，通常希望不同节点间的任务关联性越小越好，在 MPI 的编程实践中就是“用计算换数据通信”的原则，使系统尽可能少地进行数据交换。MPI 的消息传递机制为计算的并行化提供了灵活的方法，但目前对于任意问题的自动并行化并没有非常有效的方法，因此计算的切分工作往往需要编程人员自己根据经验来完成，这种灵活性是以增加编程的难度为代价的。

基于 MPI 的高性能计算是一种典型的面向计算的分布式系统，这种典型的面向计算的系统往往要求节点的计算能力越强越好，从而降低系统的数据通信代价。MPI 的基本工作过程可以总结为：切分计算，注入程序，启动计算，读取数据。MPI 虽然是典型的面向计算的分布式系统，但它也有类似于后来 Google 系统中的 MapReduce 能力，如 MPI 提供 MPI_Reduce（ ）函数实现 Reduce 功能，只是没有像 GFS 的分布式文件系统的支持，MPI 的 Reduce 能力是相对有限而低效的，并不能实现计算在数据存储位置发起的功能。

通常将 MPI 这样以切分计算实现分布式计算的系统称为基于计算切分的分布式计算系统。这种系统计算和存储的协作是通过存储向计算的迁移来实现的，也就是说系统先定位计算节点再将数据从集中存储设备通过网络读入计算程序所在的节点，在数据量不大时这种方法是可行的，但对于海量数据读取这种方式会很低效。

4.6.2 基于计算和数据切分的混合型分布式计算技术——网格计算

硬件和网络发展到一定阶段后，硬件价格的便宜使大多数人都有了自己的个人电脑，但却出现了一方面一些需要大量计算的任务资源不够，另一方面大量个人电脑闲置的问题。得益于网络的发展网格技术正好是在这个时期解决这一矛盾的巧妙方法。人们对网格技术的普遍理解是：将分布在世界各地的大量异构计算设备的资源整合起来，构建一个具有强大计算能力的超级计算系统。

网格计算出现于 20 世纪 90 年代，网格出现的历史背景是当时全世界已有了初步的网络，硬件价格还较昂贵，个人电脑已逐步普及，但在面对海量计算时当时的计算中心还是显得力不从心，利用世界各地闲置的计算资源构建一个超级计算资源池具有了可能性。“计算网格是提供可靠，连续、普遍、廉价的高端计算能力的软硬件基础设施”。建立于 1998 年的全球网格论坛 (GGF) 在 2006 年与企业网格联盟(EGA)合并成为开放式网格论坛(OGF)，这一组织的目标是为网格计算定义相关的开放标准。美国 Argonne 国家实验室与南加州大学信息科学学院合作开发的 Globus 工具包实现了这些标准，这个工具箱已经成为网格中间件事实上的标准。

一种典型的被大家所熟悉的网格架构如图 4.5 所示。

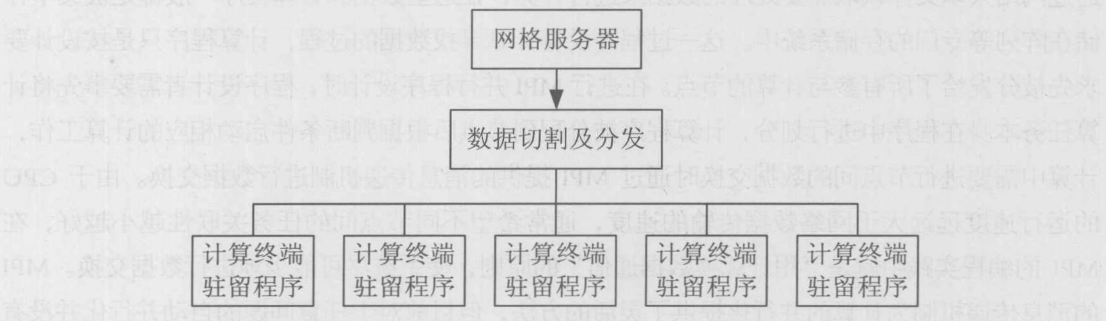


图 4.5 典型网格系统的基本架构

如图 4.5 所示的网格系统往往事先会将计算程序以某种形式安装（如屏幕保护程序）在异

构计算终端服务器上，用于监控计算终端的工作状态，当计算终端空闲时就会启动计算程序对数据进行处理，网格服务器则负责切分数据向计算终端分发数据并汇总计算结果。网格系统的数据逻辑上也是集中存储的，网格服务器负责切分数据并向计算终端传送需要计算的数据块。在这种系统结构下计算和数据的协作机制是通过数据来寻找计算实现的，即在网格中移动的主要是数据而不是计算，这种情况在数据量较小时是容易实现的，但如果需要处理的数据量很大，那么这种以迁移数据为主的方法就显得很不方便了。在网格系统中计算是先于数据到达计算终端的，这与 MPI 十分相似，数据是由计算程序主动发起请求获得，从而实现计算和数据的一致性。总体来看，网格系统既具有面向数据系统中切分数据来实现分布式计算的思想，又具有面向计算的系统中计算向数据迁移的特征，所以典型的网格系统是一种既有面向数据又有面向计算特征的混合系统，完成的任务主要还是计算密集的需要高性能计算的任务，应用领域主要是在科学计算等专业的领域。这里比较著名的网格项目就是外星文明搜索计划 SETI(Search for Extra-Terrestrial Intelligence)，应用于该计划的 SETI@home 网格将计算程序制作为屏幕保护程序借用网络上闲置的计算资源，在计算终端空闲时向网格服务器请求切分好的数据块并对该数据块进行计算，计算完成后将结果返回给网格服务器汇总。

随着面向数据逐步成为计算发展的主流，网格技术也在不断改变，Globus 也面向大数据进行了相应的改变以适应当前的实际需求，网格技术现在已呈现出全面向云计算靠拢的趋势。而作为典型的网格技术可以被认为是从面向计算走向面向数据发展过程中的过渡性技术，网格计算会在专业领域获得更好的发展但可能会在一定程度上淡出普通用户的视野，网格计算的一些思想和技术为后来云计算技术的出现提供了可以借鉴的方法。

4.6.3　基于数据切分的分布式计算技术

进入网络高速发展的时期，数据的产生成为了全民无时无刻不在进行的日常行为，数据量呈现出了爆炸式增长，大数据时代到来，数据的作用被提到很高的地位，人们对数据所能带来的知识发现表现出强烈的信心。长期以来数据挖掘技术的应用一直都处于不温不火的状态，大数据时代的到来也使这一技术迅速地被再次重视起来，基于海量数据的挖掘被很快应用于网页数据分析、客户分析、行为分析、社会分析，现在可以经常看到被准确推送到自己电脑上的产品介绍和新闻报道就是基于这类面向数据的数据挖掘技术的。基于数据切分实现分布式计算的方法在面向计算时代也被经常使用，被称为数据并行(data parallel)方法，但在面向计算时代真正的问题在于计算和数据之间只是简单的协作关系，数据和计算事实上并没有很好地融合，计算只是简单读取其需要处理的数据而已，系统并没有太多考虑数据的存储方式、网络带宽的利用率等问题。

通过数据切分实现计算的分布化是面向数据技术的一个重要特征，2003 年 Google 逐步公开了它的系统结构，Google 的文件系统 GFS 实现了在文件系统上对数据进行切分，这一点对利用 MapReduce 实现对数据的自动分布式计算非常重要，文件系统自身就对文件施行了自动的切分完全改变了分布式计算的性质，MPI、网格计算都没有相匹配的文件系统支持，从本质上看

数据都是集中存储的，网格计算虽然有数据切分的功能，但只是在集中存储前提下的切分。具有数据切分功能的文件系统是面向数据的分布式系统的基本要求。

2004 年 Jeffrey Dean 和 Sanjay Ghemawat 发表文章描述了 Google 系统的 MapReduce 框架。与 MPI 不同，这种框架通常不是拆分计算来实现分布式处理，而是通过拆分数据来实现对大数据的分布式处理，MapReduce 框架中分布式文件系统是整个框架的基础如图 4.6 所示，这一框架下的文件系统一般将数据分为 64MB 的块进行分布式存放，需要对数据进行处理时将计算在各个块所在的节点直接发起，避免了从网络上读取数据所耗费的大量时间，实现计算主动“寻找”数据的功能，大大简化了分布式处理程序设计的难度。在这里数据块被文件系统预先切分是 MapReduce 能自动实现分布式计算的重要前提，系统通过主节点的元数据维护各数据块在系统中存储的节点位置，从而使计算能有效地找到所需要处理的数据。MapReduce 这种大块化的数据拆分策略非常适合对大数据的处理，过小的数据分块会使这一框架在进行数据处理时的效率下降。这一框架在获得良好的大数据并行处理能力的时候也有其应用的局限，MapReduce 框架在对同类型大数据块进行同类型的计算处理时具有非常好的自动分布式处理能力，但在数据较小、数据类型复杂、数据处理方式多变的应用场景效率相对低下。为了实现 Google 系统良好的计算和数据的协作机制 GFS 和 MapReduce 是密不可分的，没有 GFS 支持单独的采用 MapReduce 是没有太大价值的。

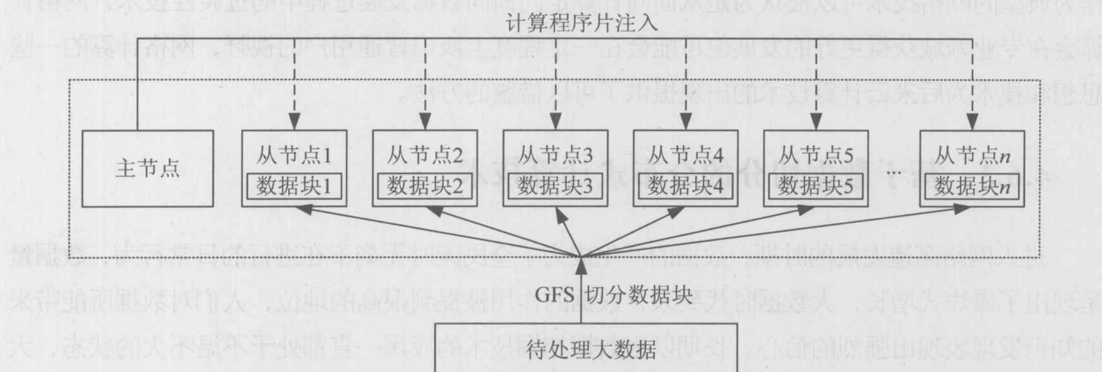


图 4.6 基于数据切分的分布式系统结构

MapReduce 框架使计算在集群节点中能准确找到所处理的数据所在节点位置的前提是所处理的数据具有相同的数据类型和处理模式，从而可以通过数据的拆分实现计算向数据的迁移，事实上这类面向数据系统的负载均衡在其对数据进行分块时就完成了，系统各节点的处理压力与该节点上的数据块的具体情况相对应，因此 MapReduce 框架下某一节点处理能力低下可能会造成系统的整体等待形成数据处理的瓶颈。在 MapReduce 框架下节点服务器主要是完成基本的计算和存储功能，因此可以采用廉价的服务器作为节点，这一变化改变了人们对传统服务器的看法。2005 年 Apache 基金会以 Google 的系统为模板启动了 Hadoop 项目，Hadoop 完整地实现了上面描述的面向数据切分的分布式计算系统，对应的文件系统为 HDFS，Hadoop 成为了面向数据系统的一个被广泛接纳的标准系统。

数据分析技术是基于数据切分的分布式系统的研究热点。对类似于 Web 海量数据的分析需要对大量的新增数据进行分析，由于 MapReduce 框架无法对以往的局部、中间计算结果进行存储，MapReduce 框架只能对新增数据后的数据集全部进行重新计算，以获得新的索引结果，这样的计算方法所需要的计算资源和耗费的计算时间会随着数据量的增加线性增加。Percolator 是一种全新的架构，可以很好地用于增量数据的处理分析，已在 Google 索引中得到应用，大大提升 Google 索引更新速度，但与 MapReduce 等非增量系统不再兼容，并且编程人员需要根据特定应用开发动态增量的算法，使算法和代码复杂度大大增加。Incoop 提出了增量 Hadoop 文件系统(Inc-HDFS)，HDFS 按照固定的块大小进文件划分，而 Inc-HDFS 则根据内容进行文件划分，当文件的内容发生变化时，只有少量的文件块发生变化，大大减少了 Map 操作量。

迭代操作是 PageRank、K-means 等 Web 数据分析的核心操作，MapReduce 作为一种通用的并行计算框架，其下一步迭代必须等待上一步迭代完成并把输出写入文件系统才能进行，如果有终止条件检查也必须等待其完成。同时，上一步迭代输出的数据写入文件系统后马上又由下一步迭代读入，导致了明显的网络带宽、I/O、CPU 时间的浪费。iHadoop 在分析了迭代过程存在的执行相关、数据相关、控制相关之后对潜在的可并行性进行了挖掘，提出了异步迭代方式，比 Hadoop 实现的 MapReduce 执行时间平均减少了 25%。Twister 对 MapReduce 的任务复用、数据缓存、迭代结束条件判断等进行调整以适合迭代计算，但其容错机制还很欠缺。

Pregel 是 Google 提出专用于解决分布式大规模图计算的计算模型，非常适合计算如 FaceBook 等社交关系图分析，其将处理对象看成是连通图，而 MapReduce 将处理对象看成是 Key-Value 对；Pregel 将计算细化到顶点，而 MapReduce 将计算进行批量化，按任务进行循环迭代控制。

在分布式文件系统条件下数据的切分使对文件的管理变复杂，此类集群系统下文件系统的管理和数据分析是需要进行重点关注的研究领域之一。

4.6.4 三种分布式系统的分析对比

从面向计算发展到面向数据，分布式系统的主要特征也发生了变化，表 4.2 中对 3 种典型的分布式系统进行了对比和分析，从表中可以看出分布式系统的发展大体分为了三种类型：面向计算的分布式系统、混合型分布式系统和面向数据的分布式系统。其中混合型分布式系统是发展过程中的一个中间阶段，它同时具有面向计算和面向数据的特征，如混合型系统中也存在数据拆分这类面向数据系统的典型特征，但却是以集中式的存储和数据向计算迁移的方式实现计算和数据的位置一致性。对于面向数据的分布式系统往往有对应的分布式文件系统的支持，从文件存储开始就实现数据块的划分，为数据分析时实现自动分布式计算提供了可能，计算和数据的协作机制在面向数据的系统中成为了核心问题，其重要性凸显出来。

表 4.2　3 种分布式系统的对比

	面向计算的分布式系统	混合型分布式系统	面向数据的分布式系统
分布式计算的实现方法	计算拆分	数据拆分	数据拆分
典型的存储方式	集中存储	集中存储	分布式存储
计算与数据的位置一致性关系	数据向计算迁移	数据向计算迁移	计算向数据迁移
系统物理位置模式	集中	分散	集中
节点性能要求	高	低	中
计算与数据协作机制	计算直接读取数据	计算主动请求读取数据	一致性哈希，主节点元数据
并行程序开发难度	难	N/A	易
应用场景	计算密集	计算密集	数据密集
负载均衡方式	CPU 参数均衡	CPU 参数均衡，数据块均衡	数据块均衡
主要应用领域	专业领域	专业领域	普通领域
典型系统	MPI，高性能计算	网格计算，高性能计算	Hadoop、Dyname、Cassandra、Google

由于面向计算的分布式系统具有灵活和功能强大的计算能力，能完成大多数问题的计算任务，而面向数据的分布式系统虽然能较好地解决海量数据的自动分布式处理问题，但目前其仍是一种功能受限的分布式计算系统，并不能灵活地适应大多数的计算任务，因此现在已有一些研究工作在探讨将面向计算的分布式系统与面向数据分布式系统进行结合，希望能在计算的灵活性和对海量数据的处理上都获得良好的性能。目前技术的发展正在使面向计算和面向数据的系统之间的界限越来越不明确，很难准确地说某一个系统一定是面向计算的还是面向数据的系统，数据以及面向数据的计算在云计算和大数据时代到来时已紧密结合在了一起，计算和数据的协作机制问题也成为重要的研究课题。

特别是 HPCC（High Performance Computing Cluster，高性的计算集群）系统的出现表明这一融合过程正在成为现实，HPCC 系统是 LexisNexis 公司开发的面向数据的开源高性能计算平台，HPCC 的详细内容参见本书第 7 章。HPCC 采用商品化的服务器构建的面向大数据的高性能计算系统，HPCC 系统希望能结合面向数据和面向计算系统的优点，既能解决大数据的分布式存储问题又能解决面向大数据的数据处理问题。

HPCC 系统主要由数据提取集群 Thor，数据发布集群 Roxie 和并行编程语言 ECL(Enterprise Control Language)组成。其中 Thor 集群是一个主从式集群，这一集群有一个能实现冗余功能的

分布式文件系统 Thor DFS 支持，主要完成大数据的分析处理，从类比的角度可以将这一部分看成是一个有分布式文件系统支持的 MPI，这一点正好弥补了 MPI 没有分布式文件系统支持的弱点。在 HPCC 系统中高性能计算和大数据存储的融合再次提示：计算和数据的协作问题是解决面向数据时代大数据分析处理问题中的一项关键技术。

练习题

1. 云计算技术领域存在两个主要技术路线，一个是＿＿＿＿＿＿＿＿＿＿，另一个是＿＿＿＿＿＿＿＿＿＿。

2. 集群的设计中要考虑 5 个关键问题是＿＿＿＿、＿＿＿＿、＿＿＿＿、＿＿＿＿、＿＿＿＿。

3. 传统的集群系统可以分为＿＿＿＿、＿＿＿＿、＿＿＿＿、＿＿＿＿4 类。

4. 简述 Beowulf 系统的主要特点。

5. Lustre 存储系统的组成有＿＿＿＿、＿＿＿＿、＿＿＿＿、＿＿＿＿、＿＿＿＿。

6. 简述面向计算的分布式系统、混合型分布式系统、面向数据的分布式系统的实现机制，分析三种系统的区别。

PART 5

第 5 章 MPI——面向计算的高性能集群技术

大量的云计算系统是基于集群系统的并行计算系统，了解并行计算技术，熟悉在集群条件下的工作环境是学习云计算的重要基础，MPI 为我们提供了了解集群之间通信机制的一种重要模型，从这一节开始我们进入并行计算环境，这是迈入云计算时代的第一步，熟悉在并行环境下的工作方式对我们理解云计算是有益的。

5.1 什么是 MPI

MPI (Message Passing Interface，消息传递接口)，于 1994 年 5 月标准的 1.0 版本诞生。2012 年发布了 MPI3.0，MPI 标准描述是是一种消息传递编程模型，并成为这种编程模型的代表和事实上的标准，MPI 本身并不是一个具体的实现，而只是一种标准描述，MPI 库可以被 FORTRAN77/C/Fortran90/C++调用。消息传递机制使服务器之间能有机的结合在一起形成一个更大的计算资源池，通过消息通信机制服务器之间能进行数据交换从而实现对计算任务的相互协作。目前在高性能计算领域 MPI 也是事实上的标准，许多超级计算机上都安装了符合 MPI 标准的软件平台。大量的计算软件也是基于 MPI 完成的，如也纳大学 Hafner 小组开发的进行电子结构计算和量子力学-分子动力学模拟软件包 VASP(Vienna Ab-initio Simulation Package)，它是目前材料模拟和计算物质科学研究中最流行的商用软件之一，大量的科研机构都在采用此软件，并因此诞生了一门新的学科计算材料学。MPI 标准的相关知识可参考其官方论坛 http://www.mpi-forum.org/。

MPICH 是 MPI 标准的一个最常用的开源实现，其版本基本与 MPI 标准基本同步，MPICH 的开发主要是由 Argonne National Laboratory 和 Mississippi State University 共同完成的一个 MPI 的具体实现，目前常用版本为 MPICH2（版本号 0.9-1.5），最新版本为 MPICH 3.0 。相关软件与说明书可在其官方网站免费下载：http://www.mpich.org/。

5.2 MPI 的架构和特点

云计算的定义中计算资源池的形成是十分重要的一项技术，MPI 的核心工作就是实现大量服务器计算资源的整合输出，MPI 为分布式程序设计人员提供了最大的灵活性和自由度，但随之而来的代价是编程的复杂性，程序设计人员需要自己实现任务在节点中的分配，并保证节点间的协调工作，当面对上千个节点的分布式系统时这种编程模式会成为程序员的噩梦。目前 MPI 的应用领域主要还是科学计算领域，但这种分布式计算机制却在后来的云计算系统中得到了或多或少的体现。

总体来看，MPI 具有以下的特点。

（1）程序编写灵活，功能强大。

MPI 为分布式程序设计人员提供了功能强大的消息通信函数，如阻塞通信、非阻塞通信、组通信、归约、自定义数据类型等。程序设计人员能在上面较为灵活的实现算法的并行化工作。

（2）能支持多种编程语言。

MPI 目前能支持 FORTRAN77/C/Fortran90/C++等语言的调用，能满足大多数科学计算的应用需要。

（3）MPI 对计算的支持强大，但对文件的支持较弱。

MPI 设计的初衷就是为了计算密集的任务定制的，是面向计算时代的典型技术，其对计算的支持十分强大，但 MPI 自身没有与之相融合的分布式文件系统，数据在 MPI 中的存储主要是依靠 NFS 等集中存储设备，计算时各节点需要通过网络从集中存储设备上读取数据，在面对大数据处理时网络带宽会成为其严重的瓶颈。这就是我们常说的“数据向计算迁移”，而 Hadoop 等云计算系统通常是“计算向数据迁移”，从而避免了网络瓶颈。

（4）MPI 需要程序设计人员自己实现求解问题的并行化。

MPI 并不为程序设计人员提供任何预设的程序并行化方案和模块，任何问题的并行化都需要程序设计人员自己来完成。任务切分和节点分配工作系统并不提供任何的监控系统支持，需要程序设计人员自己实现系统任务分配及负载的平衡。

（5）MPI 没有提供计算失效的处理机制。

MPI 并不为用户自动处理节点失效，如果在计算中出现节点失效问题需要重启计算任务。

（6）网络是 MPI 的主要瓶颈。

MPI 的消息传递机制是通过网络进行传输的，通常网络的数据传输速度与 CPU 计算速度相比要慢很多，大量的消息传递会大大的降低程序的计算效率，而且集群规模越大这个问题越严重，在 MPI 的编程原则中甚至有“用计算换通信”的说法，即宁愿多算也要尽可能地减少消息通信。所以在并行计算集群中往往会采用高速通信技术实现节点间的数据通信。

MPI 的这些特点使其在科学计算等专业领域得到了广泛的应用，但在云计算领域却被虚拟化技术和面向数据的分布式系统夺了风头，但其整合计算的方法是学习云计算的基础知识。

5.3 MPICH 并行环境的建立

MPICH 并行环境的建立主要就要完成以下 3 项工作。

（1）配置好 NFS 服务，实现所有节点对主节点指定文件夹的共享，该文件夹为 MPICH 的安装位置、数据和程序的存储位置，这样就可以避免在每个节点安装 MPICH，启动计算时也可以避免每次向各个节点分发程序。

（2）配置好节点间的互信，这一步就是实现集群内部各节点间无需密码访问，因为 MPICH 在计算时需要在各节点进行数据交换，集群内的节点应用相互信任的节点。

（3）编译安装配置 MPICH。

5.3.1 配置前的准备工作

我们假设集群是 4 个节点。

（1）安装 Linux 系统，并保证每个节点的 sshd 服务能正常启动。

（2）为每个节点分配 IP 地址，IP 地址最好连续分配，如 192.168. 1.1、192.168.1.2、…、192.168.1.4。

（3）配置/etc/hosts 文件，该文件可以实现 IP 地址和机器的对应解析，所有节点的该文件均要按下面的内容修改：

```
192.168.1.1 node1
192.168.1.2 node2
192.168.1.3 node3
192.168.1.4 node4
```

通过以上配置后节点之间能够通过各节点的机器名称相互访问。例如，可以通过 ping node2 或 ssh node2 进行测试。

5.3.2 挂载 NFS

由于 MPICH 的安装目录和用户可执行程序在并行计算时需要在所有节点存副本，而且目录要相互对应，每次一个节点一个节点地复制非常麻烦，采用 NFS（Network File System，网络文件系统）后可以实现所有节点内容与主节点内容同步更新，并自动实现目录的对应。NFS 使所有机器都能以同样的路径（假设为/usr/cluster）访问服务器上保存的文件，访问方法如同对本地文件的访问。这对于部分采用 MPI 进行并行计算的用户来说可能是必须的，通常我们会将 MPICH 的安装目录及并行程序存放目录配置为 NFS 共享目录，这样可以省去将文件向各个节点复制的麻烦，大大提高工作效率。

NFS 的配置方法示例如下(假设 NFS 服务器 IP 为 192.168.1.1，配置需在 root 用户下完成)。

（1）服务器端配置方法（下面的配置只在主节点进行）。

① /etc/exports 文件配置。

在文件/etc/exports 中增加以下几行:

```
/usr/cluster   192.168.1.2(rw)
/usr/cluster   192.168.1.3(rw)
/usr/cluster   192.168.1.4(rw)
```

这几行文字表明 NFS 服务器向 IP 地址为 192.168.1.2，192.168.1.3、192.168.1.4 的 3 个节点共享其/usr/cluster 目录，并使这些节点具有可读写权限。如有更多的节点可按此方法填写。

② 启动 NFS 服务。

启动 NFS 服务只需要以下两个命令：

```
service portmap start
```

注：在最新内核中，NFS 守护进程改为 rpcbind，如是新内核，启动 NFS 守护进程的命令是“service rpcbind start”。

```
service nfs start
```

到此 IP 为 192.168.1.1 的服务器已可以向其他两个节点提供/usr/cluster 目录的文件共享。

（2）客户端配置方法（需要在所有子节点做同样的配置）。

① 建立共享目录。

建立与服务器相同的共享目录用于共享服务器文件：

```
mkdir /usr/cluster
```

② 查看服务器已有的共享目录（这步可省略）。

```
showmount -e 192.168.1.1
```

通过这条命令我们可以查看 IP 地址为 192.168.1.1 的服务器可以共享的目录情况。

③ 挂载共享目录。

```
mount -t nfs 192.168.1.1:/usr /cluster  /usr/cluster
```

这一命令将 NFS 服务器 192.168.1.1 上的共享目录挂载到本地/usr/cluster 目录下。我们也可在所有子节点的/etc/fstab 文件中输入以下的代码，使文件系统在启动时实现自动挂载 NFS：

```
192.168.1.1:/usr/cluster    /usr/cluster    nfs    defaults    0 0
```

至此我们已可以实现对 NFS 共享目录的本地访问，所有子节点的/usr/cluster 文件夹都共享了 NFS 服务器的同名文件夹的内容，我们可以像访问本地文件一样访问共享文件。MPICH 的安装目录和用户存放并行程序的文件夹都需实现 NFS 共享，从而避免了每次向各节点发送程序副本。

5.3.3 配置 ssh 实现 MPI 节点间用户的无密码访问

由于 MPI 并行程序需要在各节点间进行信息传递，所以必须实现所有节点两两之间能无密码访问。节点间的无密码访问是通过配置 ssh 公钥认证来实现的。配置 ssh 是集群系统配置的常用操作，MPI、Hadoop、HPCC 系统均需配置 ssh。

例如，对新用户 user 配置 ssh 公钥认证，先在 node1 上做以下操作。

（1）生成了私钥 id_dsa 和公钥 id_dsa.pub，具体操作方法如下。

```
mkdir ~/.ssh
cd ~/.ssh
ssh-keygen -t dsa
```

系统显示如下信息，遇到系统询问直接回车即可。

```
Generating public/private dsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/user/.ssh/id_dsa.
Your public key has been saved in /home/user/.ssh/id_dsa.pub.
The key fingerprint is:
a9:8a:c7:0b:a6:59:71:03:92:ff:ac:e9:96:c2:5a:74
```

（2）将该密钥用作认证，进行访问授权。按如下命令在 node1 执行。

```
cp ~/.ssh/id_dsa.pub ~/.ssh/authorized_keys
chmod go-rwx ~/.ssh/authorized_keys
```

（3）将~/.ssh 目录下的文件复制到所有节点。

```
scp -r ~/.ssh node2:
scp -r ~/.ssh node3:
scp -r ~/.ssh node4:
```

（4）检查是否可以直接（不需要密码）登录其他节点。

```
ssh node01
ssh node02
```

如能两两之间不需密码登录其他节，则表明配置成功。

5.3.4 安装 MPICH2

MPICH2 的安装与 MPICH1 的安装有所不同，MPICH2 可以不用 machinefile。

（1）下载并解压 MPICH2 压缩包。

```
tar xvzf mpich2-1.0.tar.gz
```

解压完成后将在当前目录生成一个 MPICH 文件目录。

（2）进入 MPICH 解压后的目录，配置安装目录。

```
./configure --prefix=/usr/cluster/mpich2
```

根据以上配置 MPICH 将安装在目录/usr/cluster/mpich2，并确保所有节点已建立针对该目录的 NFS 共享。

（3）编译安装 MPICH2。进入解压后的 MPICH 文件目录，分别执行 make 和 make install 指令，这会花一段较长的时间。

（4）在当前用户目录下建立并编辑配置文件 mpd.hosts。将所有你允许访问本机进行并行计算的机器名填入，一行一个机器名，如果该机器上有两个 CPU，就将它的名字加入两次，以此类推。例如，在我们实例中的 mpd.hosts 文件内容如下：

```
node1
node2
node3
node4
```

注意，文中包含自己（即给自己放权）的目的是为了在只有一个节点时也可以模拟并行计算环境。

（5）配置环境变量。编辑 MPI 用户主目录下的~/.bashrc 文件，增加一行：

```
export PATH="$PATH:/usr/cluster/mpich2/bin"
```

这一行代码将 MPI 的安装路径加入用户的当前路径列表。重新打开命令行窗口后生效。

（6）启动 mpd 守护进程。运行 mpirun，首先要运行 mpd。在启动 mpd 守护进程前要在各个节点的用户主目录下生成一个.mpd.conf 文件，具体步骤如下：

```
touch .mpd.conf
chmod 600 .mpd.conf
mpd&
```

.mpd.conf 文件的内容为：

```
secretword=123456
```

其中，“123456”为识别口令，在所有节点中都建立该文件并保持口令一致，口令可自己设定。

mpd&为启动本地 mpd 的命令，我们也可以采用以下命令同时启动 mpd.hosts 中所列节点的 mpd。

```
mpdboot -n <节点个数> -f mpd.hosts
```

如：mpdboot -n 4 -f mpd.hosts

这一命令将同时在 mpd.hosts 文件中所指定的节点上启动 mpd 管理器。

mpd 启动后执行“mpdtrace –l”可以查看各个节点机器名。

（7）编译、运行一个简单的测试程序 cpi，这是一个 MPICH 自带的计算 π 值的并行示例程序，该例程在 MPICH 的 examples 目录下。

运行命令如下：

```
mpirun -np 4 ./cpi
```

运行结果如下：

```
Process 0 of 4 is on node0
Process 1 of 4 is on node1
```

```
Process 2 of 4 is on node2
Process 3 of 4 is on node3
pi is approximately 3.1415926544231239, Error is 0.0000000008333307
wall clock time = 0.021253
```

得到这样的输出结果表示我们所搭建的集群系统已经可以成功地运行 MPI 作业。

mpi 的编译命令为 mpicc，如编译 test.c 可用如下命令：

```
mpicc test.c
```

5.3.5 建立并行计算环境时的注意事项

第一次配置并行计算环境时，许多人都会遇到一定的困难，特别是初学者，下面我们对配置时需要注意的问题列出来，大家在遇到问题时可以参考。

（1）部分服务配置文件需要 root 用户权限，如 NFS 服务器的配置，在安装时如出现问题请首先检查当前用户是否具备相应的权限。

（2）/etc/hosts 文件需要在所有节点上修改。

（3）各节点的 MPICH 和用户程序要在相同的目录下，所有节点都必须有 MPICH 和用户程序的副本。

（4）如出现 mpd 无法启动请检查所有节点的 NFS 文件共享是否正常启动。

（5）启动 mpd 前要保证所有节点都正确安装了 MPICH。

（6）NFS 配置及取消密码配置的顺序可以交换，但 MPICH 必需最后安装，否则 MPI 不能正常运行。

（7）运行 MPI 程序时出现故障请检查以下内容：网络是否正常，各节点是否能无密码相互登录，mpd 是否在所有节点都已启动，NFS 服务及共享在各节点是否已完成，各节点.mpd.conf 文件中密码是否相同，可执行文件是否在各节点的相同路径有副本，MPICH 的安装文件是否在各节点的相同路径有副本。

（8）安装时无法采用 ssh 登录系统时请检查系统的防火墙设置。

（9）系统必须已安装 GCC 才能进行并行环境的配置。

（10）当 ssh 无密码访问配置出现故障时，可将所有节点的.ssh 文件夹全部清除，重新配置 ssh 无密码访问。

5.4 MPI 分布式程序设计基础

在并行计算时代人们更关注的是计算能力，一切以计算为中心，计算力成为了首要追逐的目标，因此人们通过将多台服务器连接起来实现计算能力的提升，这种计算模式非常适合从事计算密集型的任务，虽然云计算时代单纯的计算密集型的任务会越来越少，但了解并行计算时代的程序设计方法对我们理解云计算中的一些技术基础和理念是有好处的，我们在实际研究工

作中也体会到这点，所以本章将介绍采用 MPI 进行并行程序设计的核心技术，使读者能由并行计算世界逐步进入云计算世界。

5.4.1 最简单的并行程序

并行程序让很多人望而却步，其实基于 MPI 的并行程序设计并没有那么可怕，甚至简单得让人吃惊，本实例将介绍一段最简单的并行程序，相信读者看了这段程序之后对并行程序的畏惧将消失一大半，这段程序虽然简单，主程序只有三行，但它确实实现了多个计算节点的共同工作，也是一个真正意义上的并行程序。下面让我们从这个最简单的并行程序开始，逐渐进入并行计算世界。

1. MPI 函数说明

（1）并行初始化函数：int MPI_Init(int *argc, char ***argv)。

参数描述：argc 为变量数目，argv 为变量数组，两个参数均来自 main 函数的参数。

MPI_Init()是 MPI 程序的第一个函数调用，标志着并行程序部分的开始，它完成 MPI 程序的初始化工作，所有 MPI 程序并行部分的第一条可执行语句都是这条语句。该函数的返回值为调用成功标志。同一个程序中 MPI_Init()只能被调用一次。函数的参数为 main 函数的参数地址，所以并行程序和一般 C 语言程序不一样，它的 main 函数参数是不可缺少的，因为 MPI_Init()函数会用到 main 函数的两个参数。

（2）并行结束函数：int MPI_Finalize()。

MPI_Finalize()是并行程序并行部分的最后一个函数调用，出现该函数后表明并行程序的并行部分的结束。一旦调用该函数后，将不能再调用其他的 MPI 函数，此时程序将释放 MPI 的数据结构及操作。这条语句之后的代码仍然可以进行串行程序的运行。该函数的调用较简单，没有参数。

2. 并行源代码

程序 5.1

```
/*文件名：hello.c*/
#include "mpi.h"
#include <stdio.h>
int main(int argc,char **argv)
{
    MPI_Init(&argc,&argv); //并行部分开始
    printf("hello parallel world!\n");
    MPI_Finalize(); //并行部分结束
}
```

3. 运行结果（两个节点）

```
hello parallel world!
hello parallel world!
```

程序说明如下。

以上结果是在有两个节点的并行集群上运行的结果。本实例第一眼看上去和普通的 C 语言程序几乎一样，学过 C 语言的读者会感觉到相当的亲切和熟悉。不同的只是多了一个 mpi.h 的头文件，以及 MPI_Init()和 MPI_Finalize()这两个函数。程序虽然简单但它确实是一个真正的并行程序。程序的运行结果证明了这一点：我们程序中只有一个打印语句，按照串行程序的结果将只有一行打印结果，然而现在却出现了两行"hello parallel world!"，这是我们的两个计算节点在向你表明，我们已经进入了并行计算世界了，主程序中的打印语句被每个节点都执行了一次，我们有两个节点所以打印了两行文字，它们分别来自于不同的节点。这个实例同时揭示了并行程序的基本结构：

```
#include "mpi.h"
…
int main(int argc,char **argv) //main 函数必须带参数
{
    …
    MPI_Init(&argc,&argv); //并行部分开始
    MPI 并行程序部分
    …
    MPI_Finalize(); //并行部分结束
    …
}
```

所有并行程序都必须是这样的结构，其中 main 函数的参数 argc 和 argv 分别为程序输入参数个数及输入参数数组，MPI_Init()函数中需对 argc 和 argv 取地址&argc、&argv，这个实例是并行程序的最小应用，MPI_Init()函数和 MPI_Finalize()函数之间就是程序的并行部分，将在所有节点上获得执行，这一点读者要注意体会。MPI 的函数一般都是以 MPI_开头，所以非常容易识别。所有的 MPI 并行程序必须包含 mpi.h 头文件，因为这一头文件定义了所有的 MPI 函数及相关常数。

由于许多读者设计程序时较少用到 argc 和 argv 参数，下面的例子可以帮助大家理解它们的作用。

程序 5.2

```
/*文件名: test.c*/
#include<stdio.h>
int main(int argc,char **argv)
{
    int i;
    for(i=0;i<argc;i++)
        printf("argv[%d]=%s\n",i,argv[i]);
}
```

编译：

```
gcc test.c
```

运行命令：

```
./a.out This is main function
```

运行结果如下：

```
argv[0]=./a.out
argv[1]=This
argv[2]=is
argv[3]=main
argv[4]=function
```

这一运行结果说明了 argc 和 argv 参数的含义。

事实上在 MPI 并行运行环境下，以下串行程序的运行结果也会得到多条打印结果。

程序 5.3

```
#include <stdio.h>
int main()
{
    printf("hello parallel world!\n");
}
```

运行结果如下（两个节点）：

```
hello parallel world!
hello parallel world!
```

这是由于程序在各个节点都有备份，通过 mpirun 命令程序在所有节点都运行了一次，所以每个进程都做了一次打印操作，但在这种情况下进程没有 MPI 环境的支持，无法对自己进行识别和进行节点间的数据通信，所以无法完成真正的并行计算。只有在 MPI_Init()函数和 MPI_Finalize()函数之间的程序才具备进行消息传递模式的并行程序设计的能力，这也是我们采用 MPI 进行并行程序设计的原因。

5.4.2 获取进程标志和机器名

并行程序设计需要协调大量的计算节点参与计算，而且需要将任务分配到各个节点并实现节点间的数据和信息交换，面对成百上千的不同节点如没有有效的管理将面临计算的混乱，并行计算的实现将无法完成，因次各个进程需要对自己和其他进程进行识别和管理，每个进程都需要有一个惟一的 ID，用于并行程序解决“我是谁”的问题，从而实现对大量计算节点的管理和控制，有效地完成并行计算任务。因此获取进程标识和机器名是 MPI 需要完成的基本任务，各节点根据自己的进程 ID 判断哪些任务需要自己完成。

1. MPI 函数说明

（1）获得当前进程标识函数：int MPI_Comm_rank (MPI_Comm comm, int *rank)。

参数描述：comm 为该进程所在的通信域句柄；rank 为调用这一函数返回的进程在通信域中的标识号。

这一函数调用通过指针返回调用该函数的进程在给定的通信域中的进程标识号 rank，有了这一标识号，不同的进程就可以将自身和其他的进程区别开来，节点间的信息传递和协调均需要这一标识号。一般对于 comm 参数，我们采用 MPI_COMM_WORLD 通信域，MPI_COMM_WORLD 是 MPI 提供的一个基本通信域，在这个通信域中每个进程之间都能相互通信，我们也可建立自己的子通信域，但在这里我们使用 MPI 默认的 MPI_COMM_WORLD 通信域。

（2）获取通信域包含的进程总数函数：int MPI_Comm_size(MPI_Comm comm, int *size)。

参数描述：comm 为通信域句柄，size 为函数返回的通信域 comm 内包括的进程总数。

这一调用返回给定的通信域中所包括的进程总个数，不同的进程通过这一调用得知在给定的通信域中一共有多少个进程在并行执行。

（3）获得本进程的机器名函数：int MPI_Get_processor_name(char *name,int *resultlen)。

参数描述：name 为返回的机器名字符串，resultlen 为返回的机器名长度。

这个函数通过字符指针*name、整型指针*resultlen 返回机器名及机器名字符串的长度。MPI_MAX_PROCESSOR_NAME 为机器名字符串的最大长度，它的值为 128。

2. 并行源代码

程序 5.4

```
/*文件名：who.c*/
#include "mpi.h"
#include <stdio.h>
int main(int argc,char **argv)
{
    int myid, numprocs;
    int namelen;
```

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);//获得本进程 ID
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);//获得总的进程数目
    MPI_Get_processor_name(processor_name,&namelen);//获得本进程的机器名
    printf("Hello World! Process %d of %d on %s\n",myid, numprocs,
processor_name);
    MPI_Finalize();
}
```

3. 运行结果（3 个节点）

```
Hello World! Process 0 of 3 on wang1
Hello World! Process 1 of 3 on wang2
Hello World! Process 2 of 3 on wang3
```

4. 程序说明

本实例程序启动后会在各个节点同时执行，各节点通过 MPI_Comm_rank()函数取得自己的进程标识 myid，不同的进程执行 MPI_Comm_rank()函数后返回的值不同，如节点 0 返回的 myid 值为 0；通过 MPI_Comm_size()函数获得 MPI_COMM_WORLD 通信域中的进程总数 numprocs，通过 MPI_Get_processor_name()函数获得本进程所在的机器名。各进程调用自己的打印语句将结果打印出来，一般 MPI 中对进程的标识是从 0 开始的。在本例中机器名分别为 wang1、wang2、wang3 共 3 个节点。这里需要再次强调的是，MPI 并行程序中的变量是分布存储的，每个节点都有自己独立的存储地址空间，如 myid、numprocs、namelen 等变量在各个节点是独立的，相同的变量名它们的值是可以不同的。大家在读程序时心中一定要有变量分布存储的概念，否则将无法正确分析程序。图 5.1 解释了本实例运行时的情况：每个节点都有独立的变量存储空间，程序的副本存在于所有节点并分别得到执行，各个节点计算时的地位是平行的。

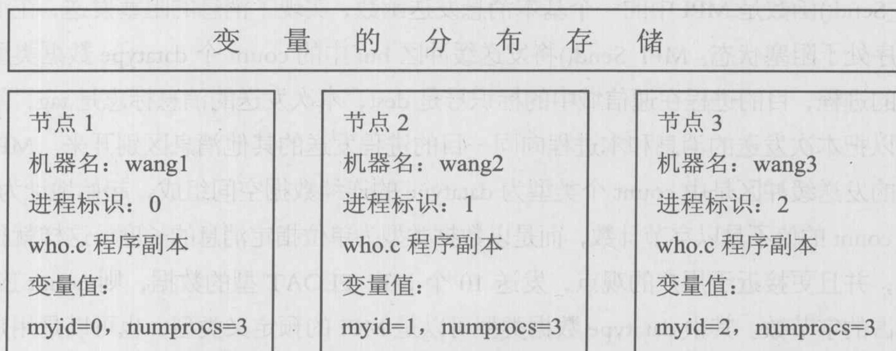


图 5.1　MPI 中变量的分布式存储方式

在3个节点的情况下如果我们的运行命令改为：

```
mpirun -np 6 ./who
```

这时会在每个节点上启动两个进程，因此程序的输出会变为：

```
Hello World! Process 0 of 6 on wang1
Hello World! Process 1 of 6 on wang2
Hello World! Process 2 of 6 on wang3
Hello World! Process 3 of 6 on wang1
Hello World! Process 4 of 6 on wang2
Hello World! Process 5 of 6 on wang3
```

这一运行结果表明程序在3个节点上启动了6个进程，每个节点都启动了两个进程。

5.4.3 有消息传递功能的并行程序

前面我们介绍的实例虽然实现了并行计算功能，但由于未采用消息传递机制，节点间由于变量地址空间是相互独立的，信息无法交换。消息传递是MPI编程的核心功能，也是基于MPI编程的设计人员需要深刻理解的功能，由于MPI的消息传递功能为我们提供了灵活方便的节点间数据交换和控制能力，掌握好MPI消息传递编程方法就掌握了MPI并行程序设计的核心。MPI为程序设计者提供了丰富的消息传递函数封装，本节的实例就是一个简单的消息传递功能的实现。

1. MPI函数说明

（1）消息发送函数 int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)。

参数描述：buf为发送缓冲区的起始地址，count将发送的数据的个数，datatype 发送数据的数据类型；dest为目的进程标识号；tag为消息标志；comm为通信域。

MPI_Send()函数是MPI中的一个基本消息发送函数，实现了消息的阻塞发送，在消息未发送完时程序处于阻塞状态。MPI_Send()将发送缓冲区buf中的count个datatype数据类型的数据发送到目的进程，目的进程在通信域中的标识号是dest，本次发送的消息标志是tag，使用这一标志就可以把本次发送的消息和本进程向同一目的进程发送的其他消息区别开来。MPI_Send()操作指定的发送缓冲区是由count个类型为datatype的连续数据空间组成，起始地址为buf。注意，这里count的值不是以字节计数，而是以数据类型为单位指定消息的长度，这样就独立于具体的实现，并且更接近于用户的观点，发送10个MPI_FLOAT型的数据，则count应为10，而不是所占的字节数。其中datatype数据类型可以是MPI的预定义类型，也可以是用户自定义的类型，但不能直接使用C语言中的数据类型。

部分C语言中的数据类型和MPI预定义的数据类型对比如表5.1所示。

表 5.1 数据类型对比

MPI 预定义数据类型	C 语言数据类型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

（2）消息接收函数：int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)。

参数描述：buf 为接收缓冲区的起始地址；count 为最多可接收的数据个数；datatype 为接收数据的数据类型；source 为接收数据的来源进程标识号；tag 为消息标识，应与相应发送操作的标识相匹配；comm 为本进程和发送进程所在的通信域；status 为返回状态。

MPI_Recv()是 MPI 中基本的消息接收函数，MPI_Recv()从指定的进程 source 接收消息，并且该消息的数据类型和消息标识和该接收进程指定的 datatype 和 tag 相一致，接收到的消息所包含的数据元素的个数最多不能超过 count 个。接收缓冲区是由 count 个类型为 datatype 的连续元素空间组成，由 datatype 指定其类型，起始地址为 buf，count 和 datatype 共同决定了接收缓冲区的大小，接收到的消息长度必须小于或等于接收缓冲区的长度，这是因为如果接收到的数据过大，MPI 没有截断，接收缓冲区会发生溢出错误，因此编程者要保证接收缓冲区的长度不小于发送数据的长度。如果一个短于接收缓冲区的消息到达，那么只有相应于这个消息的那些地址被修改，count 可以是零，这种情况下消息的数据部分是空的。其中 datatype 数据类型可以是 MPI 的预定义类型，也可以是用户自定义的类型，通过指定不同的数据类型调用 MPI_Recv()可以接收不同类型的数据。

消息接收函数和消息发送函数的参数基本是相互对应的，只是消息接收函数多了一个

status 参数，返回状态变量 status 用途很广，它是 MPI 定义的一个数据类型，使用之前需要用户为它分配空间。在 C 语言实现中，状态变量是由至少 3 个域组成的结构类型。这 3 个域分别是：MPI_SOURCE，MPI_TAG 和 MPI_ERROR。它还可以包括其他的附加域，这样通过对 status.MPI_SOURCE、status.MPI_TAG 和 status.MPI_ERROR 的引用就可以得到返回状态中所包含的发送数据进程的标识，发送数据使用的 tag 标识和该接收操作返回的错误代码。

2. 并行源代码

程序 5.5

```
/*文件名：message.c*/
#include <stdio.h>
#include "mpi.h"
int main(int argc, char** argv)
{
    int myid, numprocs, source;
    MPI_Status status;
    char message[100];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    if (myid != 0)
    {
        strcpy(message, "Hello World!");//为发送字符串赋值
        //发送字符串时长度要加 1，从而包括串结束标志
        MPI_Send(message,strlen(message)+1, MPI_CHAR, 0,99,MPI_COMM_WORLD);
    }
    else
    {
        //除 0 进程的其他进程接收来自于 0 进程的字符串数据
        for (source = 1; source < numprocs; source++)
        {
            MPI_Recv(message, 100, MPI_CHAR, source, 99,MPI_COMM_WORLD,
&status);
            printf("I am process %d. I recv string '%s' from process %d.\n",
myid, message,source);
        }
    }
```

```
    MPI_Finalize();
}
```

3. 运行结果（4 个节点）

```
I am process 0. I recv string 'Hello World!' from process 1.
I am process 0. I recv string 'Hello World!' from process 2.
I am process 0. I recv string 'Hello World!' from process 3.
```

4. 程序说明

本实例由其他进程通过 MPI 消息传递机制向 0 进程发送“Hello World”字符串数据，非 0 进程采用 MPI_Send()函数发送数据，0 进程通过循环语句分别通过 MPI_Recv()函数接收来自其他进程的字符串数据。接收缓冲区和发送缓冲区均采用同名变量 message，由于地址空间是独立的，不同进程中的 message 变量虽然名字相同但却是完全不相关的变量。程序在进行字符串的信息传递时发送长度要加 1 以包含串结束的标志。运行结果中的 3 条打印结果都是由进程 0 打印的。

MPI 的消息传递过程与信件通信的原理完全相同，如图 5.2 所示，该图将发送和接收函数中的参数与信封上的要素作了一一对应，从而帮助大家理解消息传递的机制。

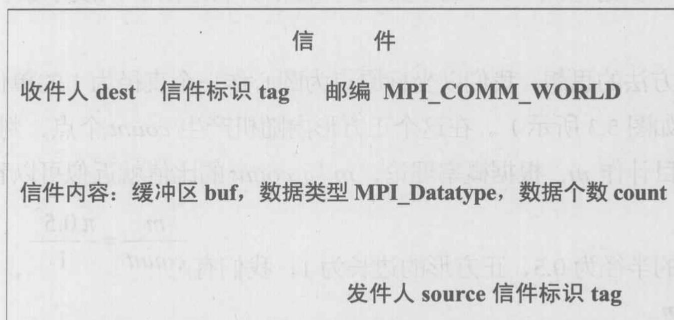


图 5.2　MPI 中消息传递与信封的对比

由于 MPI_Send()和 MPI_Recv()为阻塞通信函数，发送和接收函数一定要成对匹配，否则程序将一直处于阻塞状态无法结束。而且相关参数也要对应，如 dest 和 source 要对应，由 0 进程向 1 进程发送了信息，1 进程中一定要有一个接收函数接收来自于 0 进程的数据，而且 tag 标识要相同，因为只有相同 tag 标识的数据才能被接收函数接收。

在本例中 MPI_Send()函数和 MPI_Recv()函数实现了信息在不同节点间的传递，为节点之间协同并行工作提供了可能。我们从实例程序中看到需要 6 个 MPI 函数就能实现基本的节点间消息传递功能，这 6 个函数如下。

并行初始化函数：　　MPI_Init()

获取总的进程数函数：　MPI_Comm_size()

获取本进程 ID 函数：　MPI_Comm_rank()

消息发送函数：　　　MPI_Send()

消息接收函数：　　　MPI_Recv()

并行结束函数：　　　MPI_Finalize()

这 6 个函数是编写基于消息传递模式并行程序的最小函数集，采用这 6 个函数就能完成大多数基本的并行程序的设计，甚至有人说采用这几个函数能完成几乎所有的并行程序设计，其他函数可以用以上 6 个函数来实现，所以并行计算程序设计非常容易入门，只要掌握了这 6 个函数的用法就能编写一般的基于消息传递的 MPI 并行程序。

5.4.4 Monte Carlo 法在并行程序设计中的应用

蒙特卡罗（Monte Carlo）方法，又称随机抽样或统计试验方法，属于计算数学的一个分支，它是在 20 世纪 40 年代中期为了适应当时原子能事业的发展而发展起来的。主要思想是通过随机试验的方法，得到所要求解的问题（某种事件）出现的频率，用它们作为问题的解。简而言之，就是用频率来代替概率，当实验样本足够大的时候，就可以得到比较精确的解结果。蒙特卡罗是一种充满了魅力的算法，我们往往可以以一种简单的方法实现许多复杂的算法，大量的智能算法中也都有蒙特卡罗算法的身影，由于采用了随机数，所以蒙特卡罗方法的并行化能力特别强，而且特别简单。

本节基于蒙特卡罗思想用 MPI 程序实现对π值的并行求解，以展示蒙特卡罗算法的神奇魅力。

根据蒙特卡罗方法的思想，我们以坐标原点为圆心作一个直径为 1 的单位圆，再作一个正方形与此圆相切（如图 5.3 所示）。在这个正方形内随机产生 $count$ 个点，判断是否落在圆内，将落在圆内的点数目计作 m，根据概率理论，m 与 $count$ 的比值就近似可以看成圆和正方形的面积之比，由于圆的半径为 0.5，正方形的边长为 1，我们有 $\frac{m}{count}=\frac{\pi 0.5^2}{1}$，则$\pi$值可以用以下公式计算：$\pi=\frac{4m}{count}$。本节就采用这一方法来计算$\pi$的近似值。

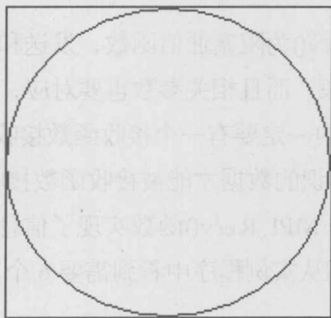


图 5.3　计算 π 值示意图

1. 并行源代码

程序 5.6

```
/*文件名: mtpi.c*/
#include"mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main(int argc,char **argv)
{
   int myid, numprocs;
   int namelen,source;
   long count=1000000;
   char processor_name[MPI_MAX_PROCESSOR_NAME];
   MPI_Status status;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);//得到当前进程的进程号
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);//得到通信域中的总进程数
   MPI_Get_processor_name(processor_name,&namelen);//得到节点主机名称
   srand((int)time(0));//设置随机种子
   double y;
   double x;
   long m=0,m1=0,i=0,p=0;
   double pi=0.0,n=0.0;
   for(i=0;i<count;i++)
   {
      x=(double)rand()/(double)RAND_MAX;//得到0~1的随机数, x坐标
      y=(double)rand()/(double)RAND_MAX;//得到0~1的随机数, y坐标
     if((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5)<0.25)//判断产生的随机点坐标是否在圆内
        m++;
   }
   n=4.0*m/1000000;
   printf("Process %d of %d on %s pi= %f\n",myid,numprocs,processor_name,n);
   if(myid!=0) //判断是否是主节点
   {
      MPI_Send(&m,1,MPI_DOUBLE,0,1,MPI_COMM_WORLD);//子节点向主节点传送结果
   }
   else
```

```
    {
        p=m;
    /*分别接收来自于不同子节点的数据*/
        for(source=1;source<numprocs;source++)
        {
            MPI_Recv(&m1,1,MPI_DOUBLE,source,1,MPI_COMM_WORLD,&status);//主节点接收
数据
            p+=m1;
        }
        printf("pi= %f\n",4.0*p/(count* numprocs));//汇总计算 pi 值
    }
    MPI_Finalize();
}
```

2. 运行结果（3 个节点）

```
Process 1 of 3 on wang2 pi= 3.141172
Process 0 of 3 on wang1 pi= 3.143284
Process 2 of 3 on wang1 pi= 3.143284
pi= 3.142580
```

3. 程序说明

本例在设计时引入 numprocs 参数，即总的节点数，通过对该参数的使用可以实现在集群节点个数发生变化时不需要对程序作任何修改，我们通常在编写并行程序时都要求能对节点的数目进行动态适应，也就是节点可扩展。在示例中各节点对落入圆内的随机点进行计数，并将计算结果发送到主节点，由主节点对所有数据汇总，并计算π值。系统打印出各节点计算的π值和汇总后的π值。这种π值计算方法收敛速度较慢，但是非常优美，随机数的威力是很强大的。这类算法具有很好的并行化能力，各节点几乎不需要作信息交换，独立完成自己的计算工作。

5.4.5 并行计算中节点间的 Reduce 操作

Map/Reduce 是 Google 引以自豪的技术之一，Map/Reduce 技术被认为能很好地实现计算的并行化，成为云计算中的一项重要技术，我们姑且不论 Map/Reduce 技术是否会成为未来云计算的主流技术，其实 Map/Reduce 也不是 Google 的创新，在 MPI 中就一直提供对各节点数据的归约（Reduce）操作，可以方便地完成多个节点向主节点的归约，并提供了相应的函数支持。云计算从技术的角度看就是从并行计算一步步走过来的。本节我们将来看看在 MPI 中是如何实现 Reduce 操作的。

本节中我们以采用 Monte-Carlo 法计算函数积分的例子来说明 MPI 中 Reduce 函数的使用方法。我们采用这一方法来计算 $y=x^2$ 在 0～10 之间的积分值。具体计算方法如图 5.4 所示。

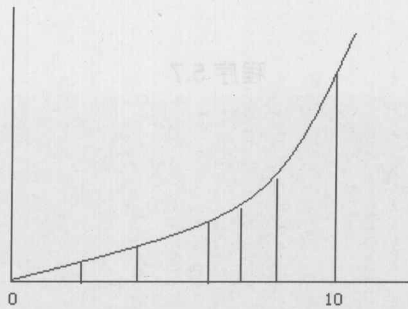


图 5.4 Monte-Carlo 计算积分的方法

该算法的思想是通过随机数把函数划分成小的矩形块，通过求矩形块的面积和来求积分值，我们生成 n 个 0～10 的随机数，求出该随机数所对应的函数值作为矩形的高，由于随机数在 n 很大时会近似平均分布于 0～10，所以矩形的宽取相同的值为 $10/n$，对所有的矩形块求和即可得函数的积分值。

1. MPI 函数说明

归约函数：int MPI_Reduce(void *sendbuf,void *recvbuf,int count,MPI_Datatype datatype, MPI_Op op,int root,MPI_Comm comm)

参数描述：sendbuf 为数据发送缓冲区；recvbuf 为数据接收缓冲区；count 为发送的数据个数；datatype 为发送的数据类型；op 为执行的归约操作；root 指定根节点；comm 为通信域。

MPI_Reduce 提供了多种归约操作，如表 5.2 所示。

表 5.2 归约操作

MPI 中的归约名	含义
MPI_MAX	求最大值
MPI_MIN	求最小值
MPI_SUM	求和
MPI_PROD	求积
MPI_LAND	逻辑与
MPI_BAND	按位与
MPI_LOR	逻辑或
MPI_BOR	按位或
MPI_LXOR	逻辑异或
MPI_BXOR	按位异或
MPI_MAXLOC	最大值且相应位置
MPI_MINLOC	最小值且相应位置

2. 并行源代码

程序 5.7

```
/*文件名 inte.c*/
#define N 100000000
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
int main(int argc, char** argv)
{
  int myid,numprocs;
  int i;
  double local=0.0;
  double inte,tmp=0.0,x;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  srand((int)time(0));//设置随机数种子
/*各节点分别计算一部分积分值*/
/*以下代码在不同节点运行的结果不同*/
  for(i=myid;i<N;i=i+numprocs)
  {
    x=10.0*rand()/(RAND_MAX+1.0);//求函数值
    tmp=x*x/N;
    local=tmp+local;//各节点计算面积和
  }
//计算总的面积和，得到积分值
 MPI_Reduce(&local,&inte,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
 if(myid==0)
 {
   printf("The integal of x*x=%16.15f\n",inte);
 }
 MPI_Finalize();
}
```

3. 运行结果

```
The integal of x*x=33.333451312647291
```

4. 程序说明

以上程序通过随机数将积分区域划分为100000000个小的区域，各节点计算一部分小矩形的面积，最后通过MPI_Reduce()函数对所有节点的计算结果进行归约求和得到最后的积分值，归约的过程就是各节点向主节点发送数据，由主节点接收数据并完成指定的计算操作，这一思想与云计算中的Map/Reduce思想类似，都是将任务分配到各节点计算最后由主节点汇总结果。程序通过myid和numpros参数的配合使同一段程序在不同的节点运行时完成不同部分的积分工作，这利用了MPI并行编程中变量分布式存储的原理，不同的节点其myid值是不同的。可见在MPI中会出现相同的代码在不同的节点执行时结果不一样的情况，这在串行程序中是不会出现的，大家要注意理解。

5.4.6 用MPI的6个基本函数实现Reduce函数功能

我们前面说过采用MPI的6个基本函数可以实现大部分的并行计算功能，但为了方便程序设计人员，MPI为我们提供了大量的函数供使用，这些功能我们大多可以采用这6个函数来实现。本节将演示如何用基本函数实现MPI中的MPI_Reduce()函数的部分功能，我们所选的问题同样是采用随机数方式计算积分，但我们并不保证MPICH一定是这样实现的。

1. 并行源代码

程序 5.8

```
/*文件名 myreduce.c*/
#define N 100000000
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
void Myreduce(double *sendbuf, double *recvbuf,int count,int root);//定义自己的reduce函数
int main(int argc, char** argv)
{
  int myid,numprocs;
  int i;
  double local=0.0;
  double inte,tmp=0.0,x;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
/*采用归约对 y=x*x 在[1,10]区间求积分*/
 srand((int)time(0));
 for(i=myid;i<N;i=i+numprocs)
 {
  x=10.0*rand()/(RAND_MAX+1.0);
  tmp=x*x/N;
  local=tmp+local;
 }
 Myreduce(&local,&inte,1,0);//调用自定义的规约函数
 if(myid==0)
 {
  printf("The integal of x*x=%16.15f\n",inte);
 }
 MPI_Finalize();
}
/*自定义的归约函数，sendbuf 为发送缓冲区，recvbuf 为接收缓冲区，count 为数据个数，root
为指定根节点*/
/*该函数实现归约求和的功能*/
void Myreduce(double *sendbuf,double *recvbuf,int count,int root)
{
 MPI_Status status;
 int i;
 int myid,numprocs;
 *recvbuf=0.0;
 MPI_Comm_rank(MPI_COMM_WORLD, &myid);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 double *tmp;
 //非 root 节点向 root 节点发送数据
 if(myid!=root)
 {
   MPI_Send(sendbuf,count,MPI_DOUBLE,root,99,MPI_COMM_WORLD);
 }
 //root 节点接收数据并对数据求和，完成规约操作
 if(myid==root)
 {
```

```
        *recvbuf=*sendbuf;
        for(i=0;i<numprocs;i++)
        {
          if(i!=root)
          {
            MPI_Recv(tmp,count,MPI_DOUBLE,i,99,MPI_COMM_WORLD,&status);
            *recvbuf=*recvbuf+*tmp;
          }
        }
    }
  }
```

2. 运行结果

```
The integal of x*x=33.332395313332192
```

3. 程序说明:

本示例程序中我们自定义了一个归约函数 Myreduce()，所用到的 MPI 函数没有超过 6 个基本函数，该函数包括 4 个参数，第一个参数 sendbuf 为各节点的发送缓冲区，第二个参数 recvbuf 为根节点的接收缓冲区，第三个参数 count 为每个节点发送的数据个数，第四个参数 root 为需要指定的接收数据并归约数据的根节点。该函数没有给出具体的归约操作，默认为对所有数据做求和的归约操作。程序执行到这一函数后将各自节点的指定数据向根节点发送,并由根节点完成求和操作,其基本的信息传递工作仍由 MPI_Send()和 MPI_Revc()函数来完成。本例我们以对函数 $y=x^2$ 在[0,10]区间求积分为例，应用自定义的 Myreduce()函数实现所有节点数据的求和，计算结果与调用 MPI_Reduce()函数的结果一样，这说明采用 6 个基本函数是可以实现大部分 MPI 功能的。我们也可以采用 6 个基本函数编写实现其他复杂的 MPI 函数。

5.4.7 设计 MPI 并行程序时的注意事项

并行程序的设计与串行程序的设计有很大的不同，需要考虑的情况更多、更复杂，因此我们编写 MPI 并行程序时要注意以下的一些问题，更多的经验需要大家在实践中进一步总结和提高。

（1）并行程序的可以执行代码和 MPICH 安装文件必须在每个节点的相同路径有副本，这可以通过复制或 NFS 共享等方法来实现。

（2）并行程序中 main 函数必须带参数，MPI_Init()函数需要这两个参数，这与一般串行程序不同。

（3）并行程序必须包含 mpi.h 这一头文件，mpi.h 中定义了 MPI 的所有函数调用及常数。

（4）并行程序中变量的地址是分布式的，各个节点的同名变量是独立的，相互没有关系，

这是要特别注意的，不要和串行程序混淆。

（5）要注意 MPI 函数调用中的参数很多都是指针，并作为函数的返回值，因此需要对变量取地址。如 MPI_Init()、MPI_Comm_rank()、MPI_Comm_size()、MPI_Send()、MPI_Recv()等。

（6）采用 MPI 进行字符串消息传递时需要将字符串的结束标志一同传送，所以字符串传输长度是字符数加 1。

（7）由于网络通信速度远低于 CPU 的技术速度，MPI 在进行消息传递时将大大增加计算时间，因此在进行 MPI 并行程序设计的时候我们往往以“计算换通信”，也就是说尽量减少节点间的数据交换，甚至可以以增加计算时间为代价。

（8）消息传递时要注意缓冲区大小的匹配，否则会出现溢出。

（9）设计并行程序时一般应能动态地适应节点个数的变化，不要因为节点个数的增减而出现修改程序源代码的问题。

（10）在进行 MPI 的非阻塞通信时，一定要在数据通信完成后才能调用接收数据的缓冲区，但可以在数据通信的同时对其他数据操作。为保证数据通信已完成，应在调用接收数据缓冲区前调用 MPI_Wait()。

（11）MPI 在进行消息传递时的数据类型一定要采用其自定义的数据类型（如 MPI_DOUBLE）或是被用户说明并提交系统的数据类型，而不能是普通的数据类型（如 double）。

练习题

1. 什么是 MPI?
2. MPI 支持________、________、________、________等语言的调用，能满足大多数科学计算的应用需要。
3. 简述 MPICH 并行环境建立的主要步骤。
4. 动手配置 MPI 节点间的 ssh 无密码访问。
5. 简述基于蒙特卡罗思想求π值的编程方法，并编写用MPI 程序。

PART 6

第6章 Hadoop——分布式大数据系统

Hadoop 是由 Apache 软件基金会研发的一种开源、高可靠、伸缩性强的分布式计算系统，主要用于对大于 1TB 的海量数据的处理。Hadoop 采用 Java 语言开发，是对 Google 的 MapReduce 核心技术的开源实现。目前 Hadoop 的核心模块包括系统 HDFS（Hadoop Distributed File System，Hadoop 分布式文件系统）和分布式计算框架 MapReduce，这一结构实现了计算和存储的高度耦合，十分有利于面向数据的系统架构，因此已成为大数据技术领域的事实标准。

Hadoop 设计时有以下的几点假设：服务器失效是正常的；存储和处理的数据是海量的；文件不会被频繁写入和修改；机柜内的数据传输速度大于机柜间的数据传输速度；海量数据的情况下移动计算比移动数据更高效。

6.1 Hadoop 概述

Hadoop 是 Apache 开源组织的分布式计算系统，其分为第一代 Hadoop 和第二代 Hadoop。第一代 Hadoop 包含 0.20.x、0.21.x、0.22.x 三个版本，0.20.x 最后演化成了 1.0.x 版本，第二代 Hadoop 包含 0.23.x 和 2.x 两个版本，2.x 本版比 0.23.x 版本增加了 NameNode HA 和 Wire-compatibility 两个特性，版本发展如图 6.1 所示。

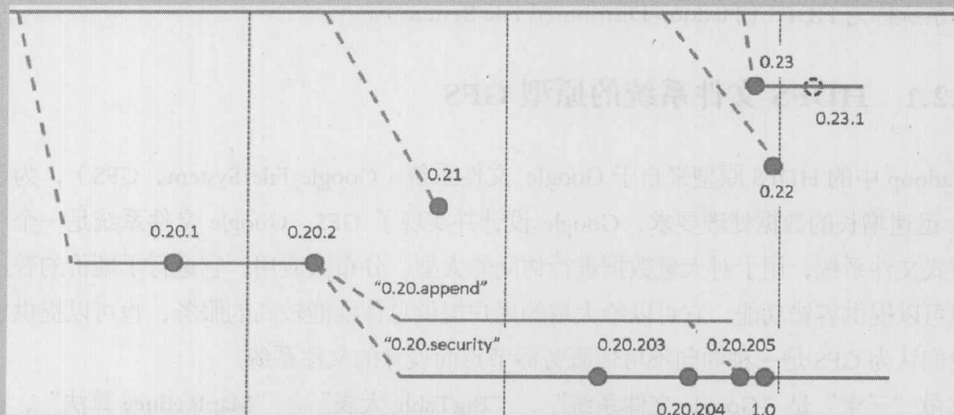


图 6.1 Hadoop 本版发展路线

Hadoop 与 MPI 在数据处理上的差异主要体现在数据存储与数据处理在系统中位置不同，

MPI 是计算与存储分离，Hadoop 是计算向存储迁移，如图 6.2 所示。



图 6.2　Hadoop 与 MPI 在数据处理上的差异

在 MPI 中数据存储的节点和数据处理的节点往往是不同的，一般在每次计算开始时 MPI 需要从数据存储节点读取需要处理的数据分配给各个计算节点对数据进行处理，因此 MPI 中数据存储和数据处理是分离的。对于计算密集型的应用 MPI 能表现出良好的性能，但对于处理 TB 级数据的数据密集型应用由于网络数据传输速度很慢，MPI 的性能会大大降低，甚至会到不可忍受的地步，所以对于构建在 MPI 上的并行计算系统网络通讯速度一直是一个重要的性能指标，用“计算换通信”也是 MPI 并行程序设计中的基本原则。

在 Hadoop 中由于有 HDFS 文件系统的支持，数据是分布式存储在各个节点的，计算时各节点读取存储在自己节点的数据进行处理，从而避免了大量数据在网络上的传递，实现“计算向存储的迁移”。

6.2　HDFS

Hadoop 系统实现对大数据的自动并行处理，是一种数据并行方法，这种方法实现自动并行处理时需要对数据进行划分，而对数据的划分在 Hadoop 系统中从数据的存储就开始了，因此文件系统是 Hadoop 系统的重要组成部分，也是 Hadoop 实现自动并行框架的基础。Hadoop 的文件系统称为 HDFS（Hadoop Distributed File System）。

6.2.1　HDFS 文件系统的原型 GFS

Hadoop 中的 HDFS 原型来自于 Google 文件系统（Google File System，GFS），为了满足 Google 迅速增长的数据处理要求，Google 设计并实现了 GFS。Google 文件系统是一个可扩展的分布式文件系统，用于对大量数据进行访问的大型、分布式应用。它运行于廉价的普通硬件上，但可以提供容错功能。它可以给大量的用户提供总体性能较高的服务，也可以提供容错功能。我们认为 GFS 是一种面向不可信服务器节点而设计的文件系统。

谷歌“三宝”是“Google 文件系统”、“BigTable 大表”、“MapReduce 算法”，有了自己的文件系统，谷歌就可以有效地组织庞大的数据、服务器和存储，并用它们工作。作为谷歌“三宝”的其中之一，GFS 的技术优势不言而喻。

GFS 为分布式结构，它是一个高度容错网络文件系统，主要由一个 Master（主）和众多

chunkserver（大块设备）构成的，体系结构如图 6.3 所示。

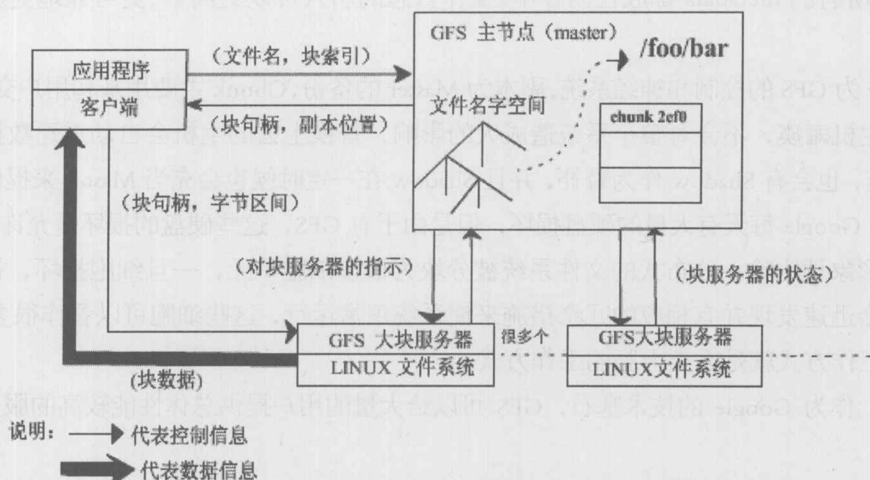


图 6.3 GFS 的体系结构

下面简单描述一下 GFS 的工作过程。

（1）客户端使用固定大小的块将应用程序指定的文件名和字节偏移转换成文件的一个块索引，向 Master 发送包含文件名和块索引的请求。

（2）Master 收到客户端发来的请求，Master 向块服务器发出指示，同时时刻监控众多 chunkserver 的状态。chunkserver 缓存 Master 从客户端收到的文件名和块索引等信息。

（3）Master 通过和 chunkserver 的交互，向客户端发送 chunk-handle 和副本位置。其中文件被分成若干个块，而每个块都是由一个不变的、全局惟一的 64 位的 chunk-handle 标识。Handle 是由 Master 在块创建时分配的。而出于安全性考虑，每一个文件块都要被复制到多个 chunkserver 上，一般默认 3 个副本。

（4）客户端向其中的一个副本发出请求，请求指定了 chunk handle（chunkserver 以 chunk handle 标识 chunk）和块内的一个字节区间。

（5）客户端从 chunkserver 获得块数据，任务完成。

通常 Client 可以在一个请求中询问多个 chunk 的地址，而 Master 也可以很快回应这些请求。

GFS 是可以被多个用户同时访问的，一般情况下，Application 和 chunkserver 是可以在同一台机子上的，主要的数据流量是通过 Application 和 chunkserver 之间，数据访问的本地性极大地减少了 Application 与 Master 之间的交互访问，减少了 Master 的负荷量，提高了文件系统的性能。

客户端从来不会从 Master 读和写文件数据。客户端只是询问 Master 它应该和哪个 chunkserver 联系。Client 在一段限定的时间内将这些信息缓存，在后续的操作中客户端直接和 chunkserver 交互。由于 Master 对于读和写的操作极少，所以极大地减小了 Master 的工作负荷，真正提高了 Master 的利用性能。

Master 保存着三类元数据（metadata）：文件名和块的名字空间、从文件到块的映射、副本位置。所有的 metadata 都放在内存中。操作日志的引入可以更简单、更可靠地更新 Master 的信息。

Master 为 GFS 的控制和神经系统，副本为 Master 的备份，Chunk 主要用来和用户交换数据。网络中的主机瘫痪，不会对整个系统造成大的影响，替换上去的主机会自动重建数据。即使 Master 瘫痪，也会有 Shadow 作为替补，并且 Shadow 在一定时候也会充当 Master 来提供控制和数据交换。Google 每天有大量的硬盘损坏，但是由于有 GFS，这些硬盘的损坏是允许的。

有人形象地比喻：分布式的文件系统被分块为很多细胞单元，一旦细胞损坏，神经系统（Master）会迅速发现并有相应的冗余措施来使系统正常运行，这些细胞可以看作很多 GFS 主机。这一工作方式就是人类大脑的工作方式。

当然，作为 Google 的技术基石，GFS 可以给大量的用户提供总体性能较高的服务，具有以下优势。

（1）Google 采用的存储方法是大量、分散的普通廉价服务器的存储方式，极大地降低了成本。

（2）对大文件数据快速存取，这个毫无疑问是可以达到的。

（3）容易扩展，它是成本很低的普通电脑，支持动态插入节点。

（4）容错能力强，它的数据同时会在多个 chunkserver 上进行备份，具有相当强的容错性。

（5）高效访问，它是通过 Big table 来实现的，它是 Google File System 上层的结构。GFS 在实现分布式文件系统的做法上面很多都是简单的，但是确实非常高效。

（6）GFS 相对于 HDFS 稳定性是无庸置疑的，并在 Google 系统中得到了采用且稳定的运行。

6.2.2　HDFS 文件的基本结构

HDFS 是一种典型的主从式的分布式文件系统，该文件系统完全是仿照 Google 的 GFS 文件系统而设计的，HDFS 的架构如图 6.4 所示。

HDFS 由一个名叫 Namenode 的主节点和多个名叫 Datanode 的子节点组成。Namenode 存储着文件系统的元数据，这些元数据包括文件系统的名字空间等，向用户映射文件系统，并负责管理文件的存储等服务，但实际的数据并不存放在 Namenode。Namenode 的作用就像是文件系统的总指挥，并向访问文件系统的客户机提供文件系统的映射，这种做法并不是 Google 或 Hadoop 的创新，这和传统并行计算系统中的单一系统映像（Single System Image）的做法相同。HDFS 中的 Datanode 用于实际对数据的存放，对 Datanode 上数据的访问并不通过 Namemode，而是与用户直接建立数据通信。Hadoop 启动后我们能看到 Namenode 和 Datanode 这两个进程。

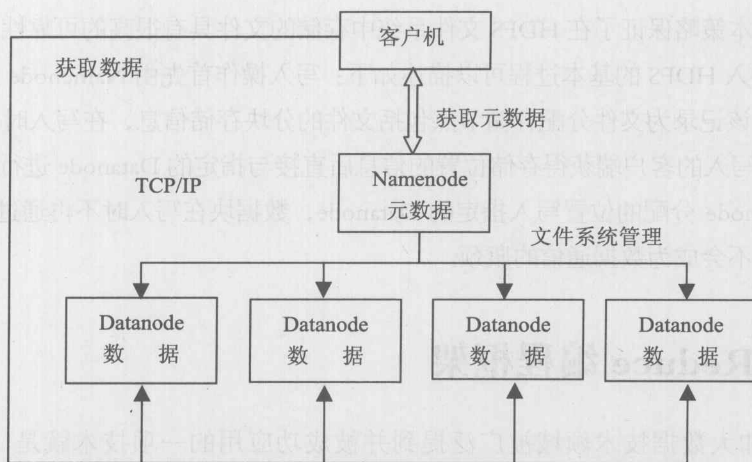


图 6.4 HDFS 的架构

HDFS 的工作过程是这样的，用户请求创建文件的指令由 Namenode 进行接收，Namenode 将存储数据的 Datanode 的 IP 返回给用户，并通知其他接收副本的 Datanode，由用户直接与 Datanode 进行数据传送。Namenode 同时存储相关的元数据。整个文件系统采用标准 TCP/IP 协议通信，实际是架设在 Linux 文件系统上的一个上层文件系统。HDFS 上的一个典型文件大小一般都在 G 字节至 T 字节。

主从式是云计算系统的一种典型架构方法，系统通过主节点屏蔽底层的复杂结构，并向用户提供方便的文件目录映射。有些改进的主从式架构可能会采用分层的主从式方法，以减轻主节点的负荷。

6.2.3 HDFS 的存储过程

HDFS 在对一个文件进行存储时有两个重要的策略：一个是副本策略，一个是分块策略。副本策略保证了文件存储的高可靠性，分块策略保证数据并发读写的效率并且是 MapReduce 实现并行数据处理的基础。

HDFS 的分块策略：通常 HDFS 在存储一个文件会将文件切为 64MB 大小的块来进行存储，数据块会被分别存储在不同的 Datanode 节点上，这一过程其实就是一种数据任务的切分过程，在后面对数据进行 MapReduce 操作时十分重要，同时数据被分块存储后在数据读写时能实现对数据的并发读写，提高数据读写效率。HDFS 采用 64MB 这样较大的文件分块策略有以下 3 个优点：

（1）降低客户端与主服务器的交互代价；

（2）降低网络负载；

（3）减少主服务器中元数据的大小。

HDFS 的副本策略：HDFS 对数据块典型的副本策略为 3 个副本，第一个副本存放在本地节点，第二个副本存放在同一个机架的另一个节点，第三个本副本存放在不同机架上的另一个

节点。这样的副本策略保证了在 HDFS 文件系统中存储的文件具有很高的可靠性。

一个文件写入 HDFS 的基本过程可以描述如下：写入操作首先由 Namenode 为该文件创建一个新的记录，该记录为文件分配存储节点包括文件的分块存储信息，在写入时系统会对文件进行分块，文件写入的客户端获得存储位置的信息后直接与指定的 Datanode 进行数据通信，将文件块按 Namenode 分配的位置写入指定的 Datanode，数据块在写入时不再通过 Namenode，因此 Namenode 不会成为数据通信的瓶颈。

6.3 MapReduce 编程框架

在云计算和大数据技术领域被广泛提到并被成功应用的一项技术就是 MapReduce。MapReduce 是 Google 系统和 Hadoop 系统中的一项核心技术。

6.3.1 MapReduce 的发展历史

MapReduce 出现的历史要追述到 1956 年，图灵奖获得者著名的人工智能专家 McCarthy 首次提出了 LISP 语言的构想，而在 LISP 语言中就包含了现在我们所采用的 MapReduce 功能。LISP 语言是一种用于人工智能领域的语言，在人工智能领域有很多的应用，LISP 在 1956 年设计时主要是希望能有效地进行"符号运算"。LISP 是一种表处理语言，其逻辑简单但结构不同于其他的高级语言。1960 年，McCarthy 更是极有预见性地提出："今后计算机将会作为公共设施提供给公众"，这一观点已与现在人们对云计算的定义极为相近了，所以我们把 McCarthy 称为"云计算之父"。MapReduce 在 McCarthy 提出时并没有考虑到其在分布式系统和大数据上会有如此大的应用前景，只是作为一种函数操作来定义的。

2004 年 Google 公司的 Dean 发表文章将 MapReduce 这一编程模型在分布式系统中的应用进行了介绍，从此 MapRuduce 分布式编程模型进入了人们的视野。可以认为分布式 MapReduce 是由 Google 公司首先提出的。Hadoop 跟进了 Google 的这一思想，可以认为 Hadoop 是一个开源版本的 Google 系统，正是由于 Hadoop 的跟进才使普通用户得以开发自己的基于 MapReduce 框架的云计算应用系统。

6.3.2 MapReduce 的基本工作过程

MapReduce 是一种处理大数据集的编程模式，它借鉴了最早出现在 LISP 语言和其他函数语言中的 map 和 reduce 操作，MapReduce 的基本过程为：用户通过 map 函数处理 key/value 对，从而产生一系列不同的 key/value 对，reduce 函数将 key 值相同的 key/value 对进行合并。现实中的很多处理任务都可以利用这一模型进行描述。通过 MapReduce 框架能实现基于数据切分的自动并行计算，大大简化了分布式编程的难度，并为在相对廉价的商品化服务器集群系统上实现大规模的数据处理提供了可能。

MapReduce 的过程其实非常简单，但上面解释看上去却较为晦涩，我们用一个实际的例子来说明 MapReduce 的编程模型。假设我们需要对一个文件 example.txt 中出现的单词次数进行统计，这就是著名的 wordcount 例子，在这个例子中 MapReduce 的编程模型可以进行如下描述。

用户需要处理的文件 example.txt 已被分为多个数据片存储在集群系统中不同的节点上了，用户先使用一个 Map 函数——Map(example.txt, 文件内容)，在这个 Map 函数中 Key 值为 example.txt，Key 通常是指一个具有唯一值的标识，value 值就是 example.txt 文件中的内容。Map 操作程序通常会被分布到存有文件 example.txt 数据片段的节点上发起，这个 Map 操作将产生一组中间 Key/value 对（word, count），这里的 word 代表出现在文件 example.txt 片段中的任一个单词，每个 Map 操作所产生的 Key/value 对只代表 example.txt 一部分内容的统计值。Reduce 函数将接收集群中不同节点 Map 函数生成的中间 Key/value 对，并将 Key 相同的 Key/value 对进行合并，在这个例子中 Reduce 函数将对所有 Key 值相同的 value 值进行求和合并，最后输出的 Key/value 对就是（word, count），其中 count 就是这个单词在文件 example.txt 中出现的总的次数。

下面我们通过一个简单的例子来讲解 MapReduce 的基本原理。

1．任务的描述

来自江苏、浙江、山东三个省的 9 所高校联合举行了一场编程大赛，每个省有 3 所高校参加，每所高校各派 5 名队员参赛，各所高校的比赛平均成绩如表 6.1 所示。

表 6.1　原始比赛成绩

江苏省		浙江省		山东省	
南京大学	90	浙江大学	95	山东大学	92
东南大学	93	浙江工业大学	84	中国海洋大学	85
河海大学	84	宁波大学	88	青岛大学	87

我们可以用如表 6.2 所示的形式来表示成绩，这样每所高校就具备了所属省份和平均分数这两个属性，即<高校名称：{所属省份，平均分数}>。

表 6.2　增加属性信息后的比赛成绩

南京大学：{江苏省，90}	东南大学：{江苏省，93}	河海大学：{江苏省，84}
浙江大学：{浙江省，95}	浙江工业大学：{浙江省，84}	宁波大学：{浙江省，88}
山东大学：{山东省，92}	中国海洋大学：{山东省，85}	青岛大学：{山东省，87}

统计各个省份高校的平均分数时，高校的名称并不是很重要，我们略去高校名称，如表 6.3 所示。

表 6.3　略去高校名称后的比赛成绩

江苏省，90	江苏省，93	江苏省，84
浙江省，95	浙江省，84	浙江省，88
山东省，92	山东省，85	山东省，87

接下来，我们对各个省份的高校的成绩进行汇总，如表 6.4 所示。

表 6.4　各省比赛成绩汇总

江苏省，90、93、84	浙江省，95、84、88	山东省，92、85、87

计算求得各省高校的平均值如表 6.5 所示。

表 6.5　各省平均成绩

江苏省，89	浙江省，89	山东省，88

以上为计算各省平均成绩的主要步骤，我们可以用 MapReduce 来实现，其详细步骤如下。

2．任务的 MapReduce 实现

MapReduce 包含 Map、Shuffle 和 Reduce 三个步骤，其中 Shuffle 由 Hadoop 自动完成，Hadoop 的使用者可以无需了解并行程序的底层实现，只需关注 Map 和 Reduce 的实现。

（1）Map Input：<高校名称，{所属省份，平均分数}>

在 Map 部分，我们需要输入<Key,Value>数据，这里 Key 是高校的名称，Value 是属性值，即所属省份和平均分数，如表 6.6 所示。

表 6.6　Map Input 数据

Key：南京大学 Value：{江苏省，90}	Key：东南大学 Value：{江苏省，93}	Key：河海大学 Value：{江苏省，84}
Key：浙江大学 Value：{浙江省，95}	Key：浙江工业大学 Value：{浙江省，84}	Key：宁波大学 Value：{浙江省，88}
Key：山东大学 Value：{山东省，92}	Key：中国海洋大学 Value：{山东省，85}	Key：青岛大学 Value：{山东省，87}

（2）Map Output：<所属省份，平均分数>

对所属省份平均分数进行重分组，去除高校名称，将所属省份变为 Key，平均分数变为 Value，如表 6.7 所示。

表 6.7 Map Output 数据

Key：江苏省 Value：90	Key：江苏省 Value：93	Key：江苏省 Value：84
Key：浙江省 Value：95	Key：浙江省 Value：84	Key：浙江省 Value：88
Key：山东省 Value：92	Key：山东省 Value：85	Key：山东省 Value：87

（3）Shuffle Output：<所属省份，List（平均分数）>

Shuffle 由 Hadoop 自动完成，其任务是实现 Map，对 Key 进行分组，用户可以获得 Value 的列表，即 List<Value>，如表 6.8 所示。

表 6.8 Shuffle Output 数据

Key：江苏省 List<Value>：90、93、84	Key：浙江省 List<Value>：95、84、88	Key：山东省 List<Value>：92、85、87

（4）Reduce Input：<所属省份，List（平均分数）>

表 6.8 中的内容将作为 Reduce 任务的输入数据，即从 Shuffle 任务中获得的（Key,List<Value>）。

（5）Reduce Output：<所属省份，平均分数>

Reduce 任务的功能是完成用户的计算逻辑，，这里的任务是计算每个省份的高校学生的比赛平均成绩，获得的最终结果如表 6.9 所示。

表 6.9 Reduce Output 数据

江苏省，89	浙江省，89	山东省，88

6.3.3 LISP 中的 MapReduce

为了进一步理解 MapReduce，我们简单介绍最早使用 Map 和 Reduce 的 LISP 语言中的 Map 和 Reduce 操作。

下面的 LISP 语句定义的这个 Map 操作是将向量（1 2 3 4 5）和向量（10 9 8 7 6）进行相乘的操作，输出也为向量（10 18 24 28 30）。

```
(map 'vector #*#(1 2 3 4 5)#(10 9 8 7 6))->#(10 18 24 28 30)
```

这个 Map 操作对应于向量到向量的映射，两个向量按乘积关系进行映射。

下面的 LISP 语句定义的这个 Reduce 操作是将向量（1 2 3 4 5 6 7 8 9 10）中的元素进行求和的 Reduce 操作,输出结果为 55。

```
(reduce #'+#(1 2 3 4 5 6 7 8 9 10))->55
```

这个 Reduce 操作对应于向量的约简，它将向量按求和的关系约简为一个值。

可以看出，在 LISP 语言中 Map 和 Reduce 只是作为一种操作定义，并没有体现出任何的分布式计算的特征。

6.3.4 MapReduce 的特点

MapReduce 主要具有以下几个特点。

（1）需要在集群条件下使用。

MapReduce 的主要作用是实现对大数据的分布式处理，其设计时的基本要求就是在大规模集群条件下的（虽然一些系统可以在单机下运行，但这种条件下只具有仿真运行的意义），Google 作为分布式 MapReduce 提出者，它本身就是世界上最大的集群系统，所以 MapReduce 天然需要在集群系统下运行才能有效。

（2）需要有相应的分布式文件系统的支持。

这里要注意的是，单独的 MapReduce 模式并不具有自动的并行性能，就像它在 LISP 语言中的表现一样，它只有与相应的分布式文件系统相结合才能完美地体现 MapReduce 这种编程框架的优势。如 Google 系统对应的分布式文件系统为 GFS，Hadoop 系统对应的分布式文件系统为 HDFS。MapReduce 能实现计算的自动并行化很大程度上是由于分布式文件系统在对文件存储时就实现了对大数据文件的切分，这种并行方法也叫数据并行方法。数据并行方法避免了对计算任务本身的人工切分，降低了编程的难度，而像 MPI 往往需要人工对计算任务进行切分，因此分布式编程难度较大。

（3）可以在商品化集群条件下运行，不需要特别的硬件支持。

和高性能计算不同，基于 MapReduce 的系统往往不需要特别的硬件支持，按 Google 的报道，他们的实验系统中的节点就是基于典型的双核 X86 的系统，配置 2~4GB 的内存，网络为百兆网和千兆网构成，存储设备的便宜的 IDE 硬盘。

（4）假设节点的失效为正常情况。

传统的服务器通常被认为是稳定的，但在服务器数量巨大或采用廉价服务的条件下，服务器的实效将变得常见，所以通常基于 MapReduce 的分布式计算系统采用了存储备份、计算备份和计算迁移等策略来应对，从而实现在单节点不稳定的情况下保持系统整个的稳定性。

（5）适合对大数据进行处理。

由于基于 MapReduce 的系统并行化是通过数据切分实现的数据并行，同时计算程序启动时需要向各节点拷贝计算程序，过小的文件在这种模式下工作反而会效率低下。Google 的实验也表明一个由 150 秒时间完成的计算任务，程序启动阶段的时间就花了 60 秒，可以想象，如果计算任务数据过小，这样的花费是不值得的，同时对过小的数据进行切分也无必要。所以 MapReduce 更适合进行大数据的处理。

（6）计算向存储迁移。

传统的高性能计算数据集中存储，计算时数据向计算节点复制，而基于 MapReduce 的分布式系统在数据存储时就实现了分布式存储，一个较大的文件会被切分成大量较小的文件存储于不同的节点，系统调度机制在启动计算时会将计算程序尽可能分发给需要处理的数据所在的节点。计算程序的大小通常会比数据文件小的多，所以迁移计算的网络代价要比迁移数据小的多。

（7）MapReduce 的计算效率会受最慢的 Map 任务影响。

由于 Reduce 操作的完成需要等待所有 Map 任务的完成，所以如果 Map 任务中有一个任务出现了延迟，则整个 MapReduce 操作将受最慢的 Map 任务的影响。

6.4 实现 Map/Reduce 的 C 语言实例

Map/Reduce 操作代表了一大类的数据处理操作方式，为了让大家对 Map/Reduce 的工作过程有一个直观的了解，下面的程序采用 C 语言实现了一个简单经典的 Map/Reduce 计算，计算从控制台输入的字符串中单词的计数，这一计算过程都是在同一个节点上完成的，并未实现计算的并行化，历史上的 Lisp 语言也是运行在单机的上的程序，这个例子的主要目的是让大家理解这一操作的过程。程序中的 my_map()和 my_reduce()函数分别实现了对字符串的 Map 和 Reduce 操作。

程序 6.1

```
/*文件名：mapreduce.c*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define BUF_SIZE        2048
int my_map(char *buffer,char (*mapbuffer)[100]);
int my_reduce(char (*mapbuffer)[100],char (*reducebuffer)[100],int *count,int num);

int main(int argc, char *argv[]){
    char buffer[BUF_SIZE];//定义存储字符串的缓冲区
    char mapbuffer[BUF_SIZE][100];//定义存储 map 结果的缓冲区
    char reducebuffer[BUF_SIZE][100];//定义存储 reduce 结果的缓冲区
    int count[BUF_SIZE]={0};//定义每个单词计数数组
    int num;//单词总数
    int i;
    int countnum;//归约后的结果数
    fgets(buffer, BUF_SIZE - 1, stdin);
```

```
        buffer[strlen(buffer)-1]='\0';//将字符串最后的回车符改为结束符
        num=my_map(buffer,mapbuffer);//调用 map 函数处理字符串
        printf("This is map results:\n");
        for(i=0;i<num;i++)
        {
            printf("<%s\t1>\n",mapbuffer[i]);
        }
        countnum=my_reduce(mapbuffer,reducebuffer,count,num);//调用 reduce 函数处理字符串
        printf("This is reduce results:\n");
        for(i=0;i<countnum;i++)
        {
            printf("<%s\t%d>\n",reducebuffer[i],count[i]);
        }

    }

    //map 函数，输入参数为字符串指针 buffer，map 后的结果通过 mapbuffer 参数传出
    //函数返回值为字符串中单词个数
    int my_map(char *buffer,char (*mapbuffer)[100])
    {


        char *p;
        int num=0;
        if(p=strtok(buffer," "))
        {
            strcpy(mapbuffer[num],p);
            num++;
        }
        else
          return num;
        while(p=strtok(NULL," "))
        {
            strcpy(mapbuffer[num],p);
            num++;
        }
        return num;
```

```
    }

    //reduce 函数，输入参数为字符串 map 后的结果 mapbuffer 和单词个数 num
    //reduce 结果通过 reducebuffer 和 count 参数传出
    //函数返回值为 reduce 的结果个数
    int my_reduce(char (*mapbuffer)[100],char (*reducebuffer)[100],int *count,int num)
    {
        int i,j;
        int flag[BUF_SIZE]={0};
        char tmp[100];
        int countnum=0;
        for(i=0;i<num;i++)
        {

           if(flag[i]==0)
           {
               strcpy(tmp,mapbuffer[i]);
               flag[i]=1;
               strcpy(reducebuffer[countnum],mapbuffer[i]);
               count[countnum]=1;
               for(j=0;j<num;j++)
               {
if(memcmp(tmp,mapbuffer[j],strlen(tmp))==0&&(strlen(tmp)==strlen(mapbuffer[j]))&&(flag[j]==0))
                   {
                        count[countnum]++;
                        flag[j]=1;
                   }
               }
               countnum++;
               }
           }
        return countnum;
    }
```

输入：

```
this is map reduce hello map hello reduce
```

输出：

```
This is map results:
<this  1>
<is    1>
<map   1>
<reduce 1>
<hello  1>
<map   1>
<hello  1>
<reduce 1>
This is reduce results:
<this  1>
<is    1>
<map   2>
<reduce 2>
<hello  2>
```

在上面的运行实例中我们从控制台输入字符串“this is map reduce hello map hello reduce”，程序通过 Map 和 Reduce 过程对字符串的的单词出现的频率进行统计，并输出了结果，这是一个典型的 Map/Reduce 过程。

6.5　建立 Hadoop 开发环境

本节使用三个 Linux 虚拟机来构建 Hadoop 集群环境，其中一个虚拟机作为 NameNode（Master 节点），另外两个虚拟机作为 DataNode（Slave 节点），如图 6.5 所示。在 3 个节点下 3 个虚拟机的机器名和 IP 地址信息如下。

虚拟机 1：主机名为 vm1，IP 为 192.168.122.101，作为 NameNode 使用；

虚拟机 2：主机名为 vm2，IP 为 192.168.122.102，作为 DataNode 使用；

虚拟机 3：主机名为 vm3，IP 为 192.168.122.103，作为 DataNode 使用。

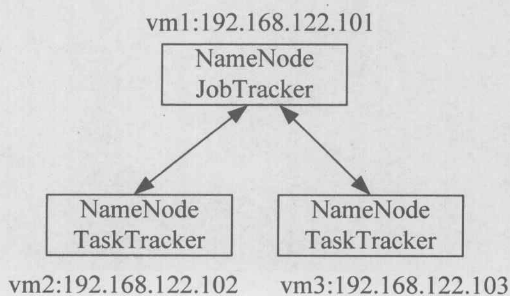


图 6.5　Hadoop 集群配置

6.5.1 相关准备工作

（1）准备虚拟机的操作系统。

首先准备3个安装了操作系统的虚拟机，本节使用的虚拟机上安装的操作系统均为CentOS 6.4（64位），先安装一机虚拟机，然后Clone两个出另外两个虚拟机。

CentOS的官方网站：www.centos.org

（2）下载Hadoop系统。

本节我们使用Hadoop的稳定版本0.21.0。

CentOS的官方网站：hadoop.apache.org

（3）下载JDK。

JDK的版本为1.7.0_45（64位）。

下载地址：www.oracle.com/technetwork/java/javase/downloads

（4）新建用户"hadoop"。

在每个节点上使用useradd指令新建一个用户hadoop，并设置密码。

```
useradd hadoop
passwd hadoop
```

（5）永久关闭每个节点的防火墙（root权限）。

在每个节点上执行以下指令，这样将永久性的关闭每个阶段的防火墙。

```
chkconfig iptables off//永久性生效，重启后不会复原
```

（6）配置ssh实现Hadoop节点间用户的无密码访问。

Hadoop需要在各个节点间进行信息传递，因此需要实现节点间的无密码访问，这与采用MPI进行并行程序设计相同，否则系统会不停地向你请求密码验证。这里的配置在各个节点的hadoop用户下进行。配置过程与MPI部分的配置过程相关，这里不再详述，配置完成后使用ssh指令可以在三个虚拟机之间实现无密码访问。

6.5.2 JDK的安装配置

Hadoop是采用Java编写的，每个虚拟机上均需要安装Java的JDK，JDK的安装配置需在root用户下进行，具体步骤如下。

（1）先在Sun官方网站下载JDK软件包jdk-7u45-linux-x64.tar.gz。

（2）新建目录/usr/java，将下载的JDK软件包进行进行解压。

```
mkdir /usr/java
tar -zxvf jdk-7u45-linux-x64.tar.gz
```

（3）配置Java环境变量。

```
#set java environment
export JAVA_HOME=/usr/java/jdk1.7.0_45
```

```
export JRE_HOME=/usr/java/jdk1.7.0_45/jre
export CLASSPATH=.:$JAVA_HOME/lib:$JRE_HOME/lib:$CLASSPATH
export PATH=$JAVA_HOME/bin:$JRE_HOME/bin:$PATH
```

（4）保存了 Java 环境变量之后，在命令行中键入如下命令，使环境变量生效：

```
source  /etc/profile          //使环境变量设置生效
```

通过 which 命令测试 JDK 的安装是否成功：

```
which java
```

系统显示如下信息：

```
/usr/java/jdk1.6.0_12/bin/java
```

此时 JDK 配置成功，接下来即可进行 Hadoop 的安装和配置。

程序 6.2

```
public class test
{
    public static void
    main(String args [ ])
    {
        System.out.println("This is a hadoop test program! JDK successed!
");
    }
}
```

编译测试程序：

```
javac test.java
```

运行测试程序：

```
java test
```

如果输出为：

```
This is a hadoop test program! JDK successed!
```

表明 JDK 已成功安装，并能成功编译执行 Java 程序。

6.5.3 下载、解压 Hadoop，配置 Hadoop 环境变量

Hadoop 集群中每个节点的安装、配置是相同的，我们可以现在一个虚拟机上进行安装、配置，然后将其复制到其他节点的相应目录。

将 hadoop-0.21.0.tar.gz 放置在 vm1 的/home/hadoop 目录中，用 hadoop 用户对其进行解压缩：

```
cd /home/hadoop
tar -zxvf hadoop-0.21.0.tar.gz
```

把 Hadoop 的安装路径添加到"/etc/profile"中，在文件的末尾添下面的代码，每个节点均需要进行此步配置。

```
#hadoop environment
export HADOOP_HOME=/usr/java/jdk1.7.0_45
export PATH=$PATH:$HADOOP_HOME/bin
```

保存了 Hadoop 环境变量之后，在命令行中键入如下命令，使环境变量生效：

```
source  /etc/profile          //使环境变量设置生效
```

6.5.4 修改 Hadoop 配置文件

Hadoop 的配置文件存于 conf 文件夹中，我们需要对该文件夹中以下文件进行修改：hadoop-env.sh、core-site.xml、mapred-site.xml、masters、slaves。

（1）修改 hadoop-env.sh 文件。

Hadoop 的 Java 环境变量在 hadoop-env.sh 中进行设置。使用 vim 打开 hadoop-env.sh 文件，找到 Java 环境变量的设置位置，将其改为 JDK 的安装地址，保存并退出。

```
export JAVA_HOME=/usr/java/jdk1.7.0_45
```

（2）修改 core-site.xml 文件。

core-site.xml 用于设置 Hadoop 集群的 HDFS 的地址和端口号，以及用于保存 HDFS 信息的 tmp 文件夹，对 HDFS 进行重新格式化的时候先行删除 tmp 中的文件。

新建 tmp 文件夹：

```
mkdir /home/hadoop/hadoop-0.21.0/tmp
```

使用 vim 打开 core-site.xml 文件，在<configuration> </configuration>之间添加以下代码：

```
<property>
       <name>hadoop.tmp.dir</name>
       <value>/home/hadoop/hadoop-0.21.0/tmp</value>
</property>
<property>
       <name>fs.defaultFS</name>
       <value>hdfs://192.168.122.101/:9000</value>
</property>
```

其中的 IP 地址需配置为集群的 NameNode（Master）节点的 IP，这里"192.168.122.101"。

（3）修改 mapred-site.xml 文件。

在 mapred-site.xml 文件的在<configuration> </configuration>之间添加以下代码，配置 JobTracker 的主机名和端口。

```
<property>
       <name>mapreduce.jobtracker.address</name>
```

```
        <value>http://192.168.122.101:9001</value>
        <description>NameNode</description>
    </property>
```

（4）修改 masters 文件。

使用 vim 打开 masters 文件，写入 NameNode（Master）节点的主机名，这里为 vm1，保存并退出。

```
vm1
```

（5）修改 slaves 文件。

使用 vim 打开 slaves 文件，写入 DataNode（Slaver）节点的主机名，这里为 vm1、vm2，保存并退出。

```
Vm2
vm3
```

6.5.5 将配置好的 Hadoop 文件复制到其他节点

到了这里，我们已经安装、配置了一个 Hadoop 节点，Hadoop 集群中每个节点的安装、配置是相同的，这里需要将 vm1 上的 Hadoop 文件夹整体复制到其他的节点，执行以下指令：

```
scp -r /home/hadoop/hadoop-0.21.0 hadoop@vm2:/home/hadoop/
scp -r /home/hadoop/hadoop-0.21.0 hadoop@vm3:/home/hadoop/
```

格式化 NameNode

在正式启动 Hadoop 之前，需要执行以下指令，对 Hadoop 的分布式文件进行初始化：

```
cd /home/hadoop/hadoop-0.21.0/bin
./hadoop namenode -format
```

顺利执行此格式化指令后，会显示如下信息：

```
    14/01/04 21:21:20 INFO common.Storage: Storage directory
/home/hadoop/hadoop-0.21.0/tmp/dfs/name has been successfully formatted.
```

6.5.6 启动、停止 Hadoop

进入/home/hadoop/hadoop-0.21.0/bin/，可以看到文件夹中有很多的启动脚本。

```
hadoop            hadoop-daemon.sh  hdfs            mapred            rcc
start-all.sh      start-dfs.sh      stop-all.sh       stop-dfs.sh
hadoop-config.sh  hadoop-daemons.sh  hdfs-config.sh  mapred-config.sh  slaves.sh
start-balancer.sh  start-mapred.sh  stop-balancer.sh  stop-mapred.sh
```

执行 start-all.sh 脚本，启动 Hadoop。

```
cd /home/hadoop/hadoop-0.21.0/bin
./ start-all.sh
```

执行之后，可以看到如下信息：

```
This script is Deprecated. Instead use start-dfs.sh and start-mapred.sh
starting namenode, logging to
/home/hadoop/hadoop-0.21.0/bin/../logs/hadoop-hadoop-namenode-vm1.out
vm2: starting datanode, logging to
/home/hadoop/hadoop-0.21.0/bin/../logs/hadoop-hadoop-datanode-vm2.out
vm3: starting datanode, logging to
/home/hadoop/hadoop-0.21.0/bin/../logs/hadoop-hadoop-datanode-vm3.out
vm1: starting secondarynamenode, logging to
/home/hadoop/hadoop-0.21.0/bin/../logs/hadoop-hadoop-secondarynamenode-vm1.out
starting jobtracker, logging to
/home/hadoop/hadoop-0.21.0/bin/../logs/hadoop-hadoop-jobtracker-vm1.out
vm2: starting tasktracker, logging to
/home/hadoop/hadoop-0.21.0/bin/../logs/hadoop-hadoop-tasktracker-vm2.out
vm3: starting tasktracker, logging to
/home/hadoop/hadoop-0.21.0/bin/../logs/hadoop-hadoop-tasktracker-vm3.out
```

从中我们可以看出，此脚本启动了 NameNode、SecondaryName、JobTracker、两个 DataNode 以及两个 TaskTracker。

在 NameNode(192.168.122.101) 上输入 jps 命令查看启动进程情况：

```
11850 SecondaryNameNode
11650 NameNode
11949 JobTracker
12132 Jps
```

在 DataNode(192.168.122.102)、DataNode(192.168.122.102) 上输入 jps 命令查看启动进程情况：

```
8727 DataNode
8819 TaskTracker
8958 Jps
```

到此 Hadoop 已经配置成功，不同的 Hadoop 版本配置方法会有所不同。

Hadoop 的停止命令如下：

```
bin/stop-all.sh
```

6.5.7 在 Hadoop 系统上运行测试程序 WordCount

（1）先在 Hadoop 用户当前目录下新建文件夹 WordCount，在其中建立两个测试文件 file1.txt，file1.txt。自行在两个文件中填写内容。

file1.txt 文件内容为

```
This is the first hadoop test program!
```

file2.txt 文件内容为

```
This program is not very difficult,but this program  is a common hadoop program!
```

（2）在 Hadoop 文件系统上新建文件夹“input”，并查看其中的内容：

```
hadoop fs -mkdir input
hadoop fs -ls
```

（3）将 WordCount 文件夹中 file1.txt、file2.txt 文件上传到刚刚创建的“input”文件夹：

```
hadoop fs -put /home/hadoop/WordCount/*.txt input
```

（4）运行 Hadoop 的示例程序 wordcount，运行命令如下：

```
    hadoop jar /home/hadoop/hadoop-0.21.0/hadoop-mapred-examples-0.21.0.jar
wordcount intput output
```

（5）查看输出结果的文件位置和 WordCount 的结果：

```
hadoop fs -ls output
```

随后出现的信息显示了输出结果的文件位置：

```
    -rw-r--r--   3 hadoop supergroup        118 2014-01-04 23:18
/user/hadoop/output/part-r-00000
```

使用如下指令查看 WordCount 的结果：

```
hadoop fs -cat /user/hadoop/output/part-r-00000
```

显示信息如下：

```
This    2
a       1
common  1
difficult,but   1
first   1
hadoop  2
is      3
not     1
program 2
program!        2
test    1
the     1
this    1
very    1
```

练习题

1. 谷歌“三宝”是__________、__________、__________。
2. 简述 GFS 的工作过程。
3. 简述 HDFS 的分块策略。
4. 简述搭建 Hadoop 开发环境的流程，并动手搭建 3 个节点的 Hadoop 集群。

PART 7

第7章 HPCC——面向数据的高性能计算集群系统

大数据时代的应用需要对采集、存储的超大规模数据进行分析处理，传统并行数据库系统无法提供实时的高性能计算。高性能计算（HPC，High Performance Computing）一般采用超级计算和集群计算两种方式。超级计算（Super Computing）是将复杂的计算任务分配给不同的处理器进行处理；集群计算（Cluster Computing）是指利用普通服务器甚至PC构建集群用于处理海量数据集，可以通过高速网络将成千上万台服务器或PC组建计算集群，Google的Map/Reduce、Hadoop、Scope、Sector/Sphere、HPCC等都是采用这种集群计算方案，以很低的成本组建具有强大计算力的集群。

数据密集型计算不仅要存储超大规模的数据，还要对数据进行复杂的计算和分析。数据密集型计算需要处理PB级的数据，具有很高的计算复杂性和应用开发复杂性，与传统的高性能计算系统具有很大不同。将计算靠近数据是处理具有海量数据的数据密集型计算的重要原则，将计算在数据存储的位置发起可以大大降低网络传输压力、提高响应速度。

结构化数据、非结构化数据都是大数据时代处理的对象，非结构化数据的数量相对于结构化数据而言非常巨大，传统数据分析平台很难对其进行处理，是大数据时代处理的重点对象。

非结构化数据一般采用文件系统进行存储，Google的GFS文件系统和Apache开源的HDFS文件系统是主流的分布式文件系统，用于数据分析领域；Sun公司的LustreFS文件系统和开源的PVFS并行虚拟文件系统具有高扩展性和高并发I/O特性，常用于科学计算。

结构化数据主要存储于数据库和分布式表结构中，在业务数据分析领域，MySQL等传统数据库无法满足用户对存储系统的可扩展需求，Google公司的BigTable和Apache的Hbase开源等NoSQL系统使用越来越广泛；在科学计算领域主要使用基于关系数据库的SDSS（Sloan Digital Sky Survey）和开源的SciDB等科学数据库。

当前主要的数据密集型集群计算系统有Hadoop、HPCC、Storm、Apache Drill、Rapid Miner、Pentaho等。

（1）Hadoop：由Apache软件基金会发起的基于块数据切分的分布式计算平台，采用Map/Reduce编程模式，具有高吞吐量、批处理的特点。

（2）HPCC：面向数据的高性能计算平台，平台基于键/值进行分析索引，用于来解决海量数据的处理与分析。

（3）Storm：基于流处理模式的分布式实时计算平台。

（4）Apache Drill：Google 大数据分析系统 Dremel 的开源版本，可在 10000 台节点上处理 PB 级的数据，常用于处理 MapReduce 产生的数据，加速 Hadoop 的查询速度，具有超低的延时。

（5）Rapid Miner：用于数据挖掘、机器学习、商业预测分析的开源计算平台。

（6）Pentaho：以工作流为核心、强调面向解决方案的商务智能软件开源项目。

计算密集型平台需要尽量减少数据的移动，大多数的超级计算将数据存储在数据仓库或者服务器，在计算的时候将数据传输到计算节点，这样的方式数据传输压力大。数据密集型计算系统通常使用分布式文件系统，将数据存储在集群节点中，在运算的时候将计算任务发送到需要处理的数据所在的节点，数据传输压力远低于超级计算模式。

大量数据节点组成的计算集群中软硬件故障、通信故障是经常出现的，系统在软硬件平台设计的时候需要考虑系统的稳定性和可用性。

7.1 HPCC 简介

当今，许多组织和企业都对数据密集型计算有着巨大的需求。2011 年，LexisNexis 公司开源了其高性能计算分析平台 HPCC 系统，其 C++编写的天然速度优势，可靠性与强力的错误恢复机制，强大易用的 ECL 编程语言模式等新特性给我们解决大数据处理问题带来了新的思路与方法。Hadoop 系统在文件分割时是基于数据块的而 HPPC 在文件分割时是基于记录的，相比 Hadoop 系统 HPCC 为用户更近一步地隐藏了分布式计算的细节，简化了并行程序的编写难度，HPCC 的社区版的开源为分布式大数据处理系统提供了一个新的选择，同时由于 HPCC 由专门的公司在支持，其安全性和稳定性能得到保证。

LexisNexis 是世界最大出版商 Elsevier 集团的子公司，其提供的 LexisNexis 数据库是著名的数据库，为全球众多政府部门、金融公司、法务部门、高科技公司等所使用，该数据库连接至 40 亿个文件、11439 个数据库以及 36000 个来源，资料每日更新。如此规模巨大的数据量对 LexisNexis 的服务系统提出了挑战。为此 LexisNexis 开发了 HPCC 高性能集群计算平台，来解决海量数据的处理与分析问题。面对日益增长的大数据应用需求，LexisNexis 公司于 2011 年推出了 HPCC 的社区开源版本，数十年来 HPCC 稳定、高效的特性保证了 LexisNexis 公司数据库正常的服务功能，开源版本的推出使其成为数据密集型计算的重要平台。同年，亚马逊公司宣布在其云计算平台上使用 HPCC 系统，这无疑为 HPCC 的发展注入了强大的动力。

HPCC 系统相对于现今的各种大数据解决方案有以下优点：

（1）强大灵活的 ECL 语言，显著提升了程序员编程的效率；

（2）HPCC 系统提供的 Roxie 集群提供了高效的在线查询和分析服务；

（3）ECL 程序首先编译为优化的 C++，高速性能得到保证；

（4）高效的错误恢复和冗余备份机制；

（5）稳定和可靠的系统；

（6）相对其他平台，在较低的系统消耗上实现了更高的性能。

HPCC在集群计算方面具备非常强大的性能，“Terabyte Sort”是著名的计算机平台性能测试的项目（http://sortbenchmark.org），测试的内容是对由10000000条100个字节的记录构成的数据集合进行排序。HPCC的官方文件表明，1460台部署了Hadoop的集群花费62秒完成1TB数据排序，而400台部署了HPCC系统的集群仅花费102秒即完成1TB数据排序。

高性能计算目前可以分为两类：一类是面向计算的高性能计算；另一类是面向数据的高性能计算。HPCC等大数据系统就是面向数据的高性能计算。HPCC集群和高性能计算中的Beowulf集群看上去非常相似，但历史上Beowulf集群的应用目标主要是针对计算，而HPCC集群的设计目的的面向海量数据处理。运行上Beowulf集群上的软件主要是MPI等，这类软件需要用户自己操作集群中的每一个节点，HPCC由于采用了数据并行的方式，在分布式文件系统的支持下可以将并行处理进程向用户隐藏，从而大大降低了分布式程序设计的复杂性。

HPCC系统分为商业版和社区版。在2011年以前，HPCC系统是一个商业系统，拥有稳定的用户群体，随着云计算、移动互联网的发展，大数据应用需求激增，开源的大数据处理平台Hadoop的普及给HPCC商业版的发展带来不小的压力，LexisNexis公司于2011年推出了开放源码的HPCC社区版，本章的内容均是基于HPCC社区版完成。社区版与商业版在大多数模块上是完全相同的，社区版能实现绝大多数的HPCC功能，HPCC社区版与商业版的主要区别如表7.1所示。

表 7.1　HPCC 社区版与商业版的区别

	社区版	商业版
概述	免费，开放源代码，可以获取社区支持，适合用于非关键型的应用	根据安装节点数收费，开放源代码，适合需要数据密集型计算平台的企业级用户
许可证	Apache License, Version 2.0	商业许可
数据加密	不支持	支持
ECL 库	ECL 基本库	ECL 通用库和附加模块
Infiniband	IPoIB	Native/RDMA
远程存储支持	不支持	支持
支持	开源社区、在线文档帮助，免费学习视频	商业咨询，Bug 定位，商业培训

7.2 HPCC 的系统架构

HPCC 系统从总体物理上可以看作在同一个集群上部署了 Thor（数据加工处理平台）和 Roxie（数据查询、分析和数据仓库）的集群计算系统，并包含 ECL 中间件、外部通信层、客户端接口和辅助组件，系统架构如图 7.1 所示。

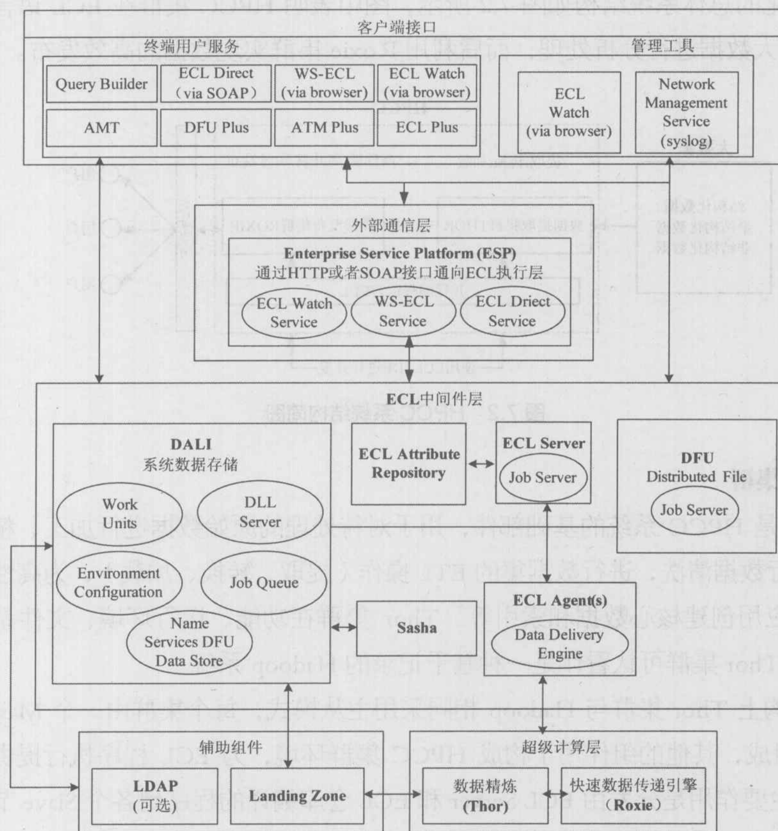


图 7.1 HPCC 系统架构

Thor 集群和 Roxie 集群是 HPCC 系统的核心部件，这两个部件可以根据并行处理任务进行独立优化。Thor 集群的任务可以独立执行任务，不需要部署 Roxie 集群；但要运行 Roxie 集群上的任务必须首先部署 Thor 集群，为其构建分布式索引文件。其中 ESP 服务器（Enterprise Service Platform）提供与用户交互的网络连接，后面用到的 ECL Watch 和 WS ECL Service 都属于 ESP 服务。

HPCC 的系统服务器包含 ECL 服务器、Dali 服务器、Sasha 服务器、DFU 服务器和 ESP 服务器，这些服务器为 Thor、Roxie 集群和外部建立接口，并为 HPCC 环境提供服务支持。

- ECL 服务器包含 ECL 编译器和执行代码生成器，是 Thor 集群的任务服务器。
- Dali 服务器的功能是数据仓库，主要用于管理工作单元数据，维护 DFU 的逻辑文件目录

信息，配置 HPCC 环境，维护系统消息队列等。

- Sasha 服务器的主要功能是尽量减轻 Dali 服务器的压力和资源利用率。
- DFU 服务器用于向 Thor 集群的分布式文件系统 DFS 发送数据和回收数据。
- ESP 服务器是外部客户端链接到集群的接口。

同时 HPCC 平台为数据分析人员、编程人员、管理人员和终端用户提供了一系列开发工具和组件，包括集成开发环境 QueryBuilder、集群监控管理工具 ECL Watch 等。

HPCC 简化的总体系统结构如图 7.2 所示，图中表明 HPCC 集群在 ECL 语言的基础上利用 Thor 集群对大数据进行分析处理，而后利用 Roxie 集群实现数据的高效发布。

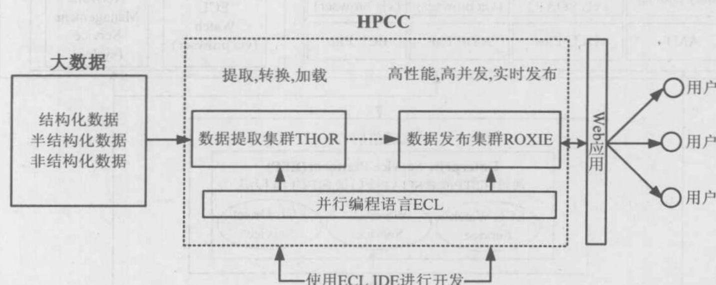


图 7.2　HPCC 系统结构简图

1．Thor 集群

Thor 集群是 HPCC 系统的基础部件，用于对待处理的原始数据进行加工、精炼，例如：对原始数据进行数据清洗，进行数据集的 ETL 操作（提取、转换、加载），为高性能结构化查询和数据仓库应用创建核心数据和索引等。Thor 集群在功能、运行环境、文件系统等方面与 Hadoop 相似。Thor 集群可认看作是一种基于记录的 Hadoop 系统。

在系统结构上 Thor 集群与 Hadoop 相同采用主从模式，每个集群由一个 Master 节点和多个 Slave 节点组成，其他的组件用于构成 HPCC 集群环境，为 ECL 程序执行提供并行环境。Master 节点的主要作用是分发由 ECL Server 和 ECL 仓库编译的程序到各个 Slave 节点，并监控、协调程序在各个 Slave 节点上的执行（类似于 Hadoop 中的 Job Tracker）。在一个 HPCC 系统中，可以建立多个 Thor 集群。

Thor 集群每一个 Slave 节点也作为集群分布式文件系统的数据节点存在，与 Map/Reduce 集群所用的块格式不同，Thor 中的数据是面向记录的，数据记录既可以是长度固定的内容也可以是其他的固定格式数据。Thor 在本地节点和集群内其他节点都提供了备份副本，每当有新数据添加，都会自动进行备份。Thor 集群上执行的任务也可以通过分布式文件系统从其他集群导入文件。Thor 集群中的分布式文件系统是 HPCC 实现大数据处理的关键，通过数据在系统中预先切分并别存储于各个子节点为后面对数据的并行化处理提供方便。正是利用分布式文件系统 HPCC 可以向开发者隐藏并行数据处理的复杂性实现算法的自动并行化，这是当前大数据系统架构采用的主要方法。

2．Roxie 集群

作为数据快速交付引擎的 Roxie(Rapid Online XML Inquiry Engine)是一个高性能的结构化查询和分析平台，支持数高并发数据请求，快速响应请求。

Roxie 提供了高性能的在线结构化数据查询和分析的功能和数据仓库的功能。其作用类似 Hadoop 中的 Hive 和 HBase，但 Roxie 的效率更高。Roxie 集群的每个节点都同时运行着 Srver 进程和 Worker 进程。Server 进程主要负责等待接收查询请求，并调度执行请求。Roxie 集群有自己的分布式文件系统，以索引为基础，使用分布式 B+树索引文件。Roxie 集群提供了强大的错误恢复和冗余备份功能，在两个或更多的节点上进行数据冗余，能在节点失效的情况下继续运行。

Roxie 集群的每个节点都有 Server 进程和 Worker 进程。Server 进程等待外部查询请求，根据查询请求确定需要的节点。Roxie 集群中的辅助组件节点是一台 ESP 服务器，外部终端通过 ESP 服务器与集群进行连接，其余组件与为 Roxie 集群创建分布式索引文件的 Thor 集群共享。

在 HPCC 编程环境中，Thor 和 Roxie 这两种并行数据处理平台需要各自优化并相互配合，HPCC 平台需要根据系统性能要求和用户的需求来决定使用两种平台各自的节点数，以获取最优性能。

Hadoop 平台在进行大数据处理时需要同时部署 HBase、Hive、Pig 等系统，这些系统的目标和需求并不相同，不能完全符合 Map/Reduce 的模式。HPCC 平台集成了 Thor 和 Roxie，可以根据实际需求配置 Thor 和 Roxie 集群，比 Hadoop 具有更大的灵活性。通常 Thor 和 Roxie 集群在物理上是同一个集群，只是完成的任务不同，它们基于的底层分布式文件系统是同一个文件系统。

7.3 HPCC 平台数据检索任务的执行过程

HPCC 的平台上的数据检索任务在 Thor 集群和 Roxie 集群上运行，执行过程包含导入原始数据、数据切分与分发、ETL 处理、Roxie 集群发布，如图 7.3 所示。

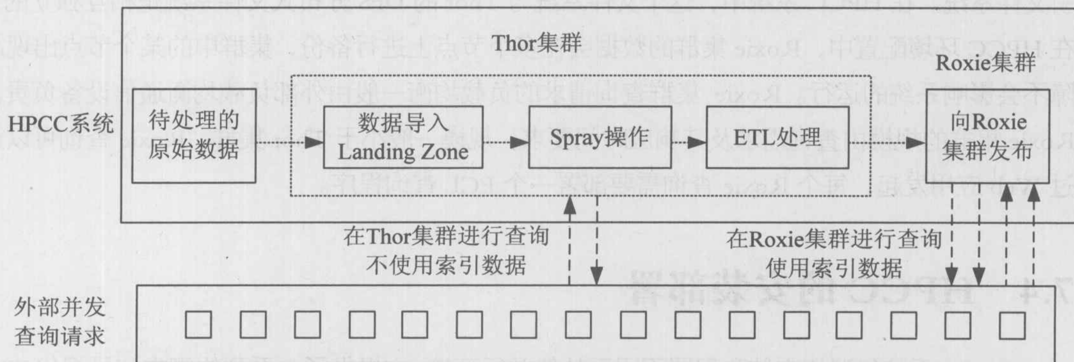


图 7.3 HPCC 数据检索任务的执行流程

1. 加载原始数据

将存储在HPCC平台以外的待处理数据加载到Thor集群，存放位置为Landing Zone，可在HPCC配置中进行查询。常用的数据加载方式有两种，一种是登录ECL Watch，通过Web方式将数据导入Thor集群；另一种是直接使用WinSCP等文件传输工具将输入传入相应节点的文件夹。

2. 切分、分发待处理的数据

这个操作对应于图7.3中的Spray，Spray操作将Landing Zone中的数据进行均匀切分，发送到Thor集群的计算节点。切分的时候根据文件的逻辑记录结构进行切分，保证逻辑记录的完整性，不被切分到多个节点上。这个操作由DFU服务器控制，将数据切分并分发到各个存储节点，切分后的文件形成一个逻辑文件，供ECL编程时使用。

在数据的处理过程中，使用ECL编程语言可以将DFS中的文件进行重新分配。例如，可以将匹配某一个键值的数据分配到一个节点上，这样对这些数据的操作处理就在一个节点上进行，避免节点间的数据移动，达到数据并行操作、性能最优。

由于HPCC中分布式文件系统是与系统紧密结合的，所以系统能有效地识别分布在不同节点上被切分后的文件，这样系统在进行数据处理时就能同时在多个节点发起，从而实现对大数据的分布式处理。同时系统所生成的索引文件在系统中也是以分布式的形式存放在系统中的不同节点。

3. 分发后原始数据的ETL处理

对分发后的原始数据ETL是Thor集群的典型应用，包含Extract操作、Transform操作和Load操作。Extract操作包含源数据映射、数据清洗、数据分析统计等操作；Transform操作是对数据集的常规操作，如数据记录的合并或拆分，数据集内容的更新，格式的变化等；Load操作的主要作用是为数据仓库或一些独立的查询平台建立索引，索引建立后会被加载到Roxie平台以支持在线查询。

4. 向Roxie集群发布

当一个查询被部署到Roxie集群，相关的支撑数据、索引文件也被加载到Roxie分布式索引文件系统。在HPCC系统中，这个文件系统与Thor的DFS分布式文件系统是相互独立的。在HPCC环境配置中，Roxie集群的数据会在多个节点上进行备份，集群中的某个节点出现故障不会影响系统的运行。Roxie集群查询请求的负载均衡一般由外部负载均衡通信设备负责。Roxie集群的规模由查询需求及其响应时间要求，规模一般小于Thor集群。Roxie查询可以通过Web应用发起，每个Roxie查询需要部署一个ECL查询程序。

7.4 HPCC的安装部署

HPCC系统集群的安装和配置不同于其他并行环境，它提供了一系列的脚本和可视化Web界面来辅助HPCC系统集群的部署。在HPCC系统的部署中，我们首先要在集群的节点间配置ssh无密码访问，然后在各个节点上安装HPCC文件。之后启动HPCC系统提供的集群配置

服务，进行 HPCC 系统的集群配置。

在本次安装过程中，我们选用 5 台安装了 64 位 CentOS 6.4 的服务器作为集群节点，这些节点的 IP 地址分别为 222.18.159.122~222.18.159.126。HPCC 系统还提供了进行集群环境配置的可视化 Web 接口 8015，查看集群整体情况的 8010 接口。HPCC 系统的部署不像 Hadoop 等需要配置环境变量等，而是提供了一个可执行的安装文件，我们从官方网站下载后，可以进行直接安装。

安装步骤如下所示。

1．配置 ssh 无密码访问

在本书第 5 章的并行环境配置过程中，我们已经配置好了 ssh 无密码登录环境，这里就不详细介绍了。

2．在每个节点上分别安装 HPCC 文件并查看运行情况

（1）执行安装命令。

HPCC 官方网站提供了两种类型的 HPCC 安装文件供用户下载，插件版和不带插件版，插件版提供了 Java、Python 等语言的支持，两种版本的安装指令有所不同，这里我们采用插件版的 HPCC 系统进行安装。

在 CentOS/RedHat 系统上，插件版和不带插版件的 HPCC 系统安装命令分别如下：

```
sudo rpm -Uvh --nodeps <HPCC安装文件名>   //带插件的 HPCC 系统安装命令
sudo rpm -Uvh <HPCC安装文件名>            //不带插件的 HPCC 系统安装命令
```

（2）启动 HPCC 系统服务：

```
sudo /sbin/service hpcc-init start
```

（3）检查启动的 HPCC 服务情况：

```
sudo /sbin/service hpcc-init status
```

（4）停止 HPCC 服务：

```
sudo /sbin/service hpcc-init stop
```

安装成功后，HPCC 会将文件安装到默认的路径。其中默认的安装目录为 /opt/HPCCSystems/目录，里边存放了 HPCC 系统的核心文件和执行脚本。集群配置文件存放在/etc/HPCCSystems/目录下；日志信息等存放在/var/lib/HPCCSystems/目录中。

3．配置集群环境

（1）停止 HPCC 系统服务。

启动配置服务前，必须停止全部节点上的 HPCC 服务，否则配置服务无法启动。HPCC 系统提供了脚本用来启动或停止集群中所有节点上的服务和对所有节点的服务启动状态进行检测。

停止集群所有节点的服务：

```
sudo -u hpcc /opt/HPCCSystems/sbin/hpcc-run.sh -a hpcc-init stop
```

查看集群所有节点的服务：

```
sudo -u hpcc /opt/HPCCSystems/sbin/hpcc-run.sh -a hpcc-init status
```

启动集群所有节点的服务：

```
sudo -u hpcc /opt/HPCCSystems/sbin/hpcc-run.sh -a hpcc-init start
```

（2）启动配置服务。

HPCC 系统提供了一个 configmgr 脚本，用于启动集群环境配置的 Web 界面接口。我们通过提供的 Web 页面进行集群环境配置。这个脚本位于 /opt/HPCCSystems/sbin/目录中，我们执行这个脚本启动配置接口：

```
sudo /opt/HPCCSystems/sbin/configmgr
```

（3）启动 Web 页面配置。

通过第二步的启动配置服务，我们启动了端口为 8015 的 Web 页面的化集群环境配置界面。通过界面我们可以部署 Thor 集群和 Roxie 集群，部署 HPCC 系统的一些服务在不同的节点。

在这里，我们启动浏览器登录 http://222.18.159.122:8015 配置集群，222.18.159.122 为启动 configmgr 服务的节点 IP。

（4）创建自己的集群配置文件。

在页面中提供了不同的配置文件创建方式，我们选择带有系统指引的创建方式（Generate new environment using wizard），同时在后边空白处填写创建配置文件的名称，配置文件的名称为 new.xml，如图 7.4 所示。

图 7.4　开始 HPCC 系统配置

（5）配置集群的网络信息。

集群配置需要提供集群的网络信息，配置文件会将 Thor 集群和 Roxie 集群以及其他服务，按照用户需求分配到集群中的各个节点上去。

我们首先将集群中各节点的 IP 填入 IP 列表中。如果集群中各节点的 IP 地址连续则可以用如 222.18.159.122-126 这样的方式填写；如果 IP 地址不连续，则以节点 IP 之间可以用‘;’相隔，如 192.168.1.101;192.168.1.106 。在本实验中，集群各节点的 IP 区域是 222.18.159.122-126，如图 7.5 所示。

图 7.5　设置 IP 地址

（6）配置 Thor 集群和 Roxie 集群的节点数。

Thor 集群和 Roxie 集群是 HPCC 系统的计算数据处理集群和数据查询分析集群。我们可以根据集群中节点的数目和具体业务对 HPCC 系统的需求，对 Thor 集群和 Roxie 集群进行个性化的配置。

在本实验中，我们选用 1 个节点作为 support 节点，4 个节点作为 Thor 集群的从节点和 Roxie 集群节点，如图 7.6 所示，Thor 集群的 Master 节点会被自动分配到 support 节点。

图 7.6　集群节点设置

（7）检查配置结果。

Thor 集群和 Roxie 集群配置完成后，系统会自动化地分配各个节点，并将相应的服务分配到各个节点上，如选定 222.18.159.122 节点作为 support 节点，同时 Thor 集群的 Master 节点也被部署到这个节点上，具体的服务和集群部署情况如图 7.7 所示。

HPCC Systems

Environment summary for new.xml

Component/Esp Services	BuildSet	Net Addresses/Port
mydropzone	DropZone	222.18.159.122
myroxie	roxie	222.18.159.123,222.18.159.124,222.18.159.125,222.18.159.126
mydali	dali	222.18.159.122
mydfuserver	dfuserver	222.18.159.122
myeclccserver	eclccserver	222.18.159.122
myesp	esp	222.18.159.122
myeclagent	eclagent	222.18.159.122
myftslave	ftslave	222.18.159.122,222.18.159.123,222.18.159.124,222.18.159.125,222.18.159.126
mysasha	sasha	222.18.159.122
mydafilesrv	dafilesrv	222.18.159.122,222.18.159.123,222.18.159.124,222.18.159.125,222.18.159.126
[illegible]	[illegible]	222.18.159.122,222.18.159.1[illegible]

Cancel　Back　Finish　Advanced View

图 7.7　集群环境部署情况

在此步完成并提交之后，已经产生了一个新的 HPCC 系统集群配置文件 new.xml。

（8）用新生成的配置文件替换原始的配置文件。

原始的配置文件都是基于单节点的，我们将新生成的配置文件替换原来的配置文件，并将配置文件分发到其他节点，替换原来的配置文件。重启 HPCC 系统服务后，会按照新生成的配置文件来启动相关的服务。新生成的配置文件存放在/etc/HPCCSystems/source/目录下，真正的运行环境在/etc/HPCCsystems/目录下。

首先确保暂停所有的 HPCC 系统服务，然后将新生成的集群配置文件替换原有的配置文件：

```
sudo cp /etc/HPCCSystems/source/newEnvironment.xml
/etc/HPCCSystems/environment.xml
```

将新生成的 environment.xml 文件发送到各个节点并替换原有的 environment.xml 文件。

（9）重新启动集群。

HPCC 系统提供了一些集群控制脚本，这些脚本存放在/opt/HPCCSystems/sbin/目录下，我们可以通过执行下面的脚本启动 HPCC 集群。

```
sudo -u hpcc /opt/HPCCSystems/sbin/hpcc-run.sh -a hpcc-init start
```

（10）检查集群的启动状况。

启动集群后，各个节点会按照配置文件启动相应的服务。我们可以根据配置文件来检查相

应的服务是否启动，如果均按照配置文件启动，则 HPCC 系统集群配置成功。我们可以通过登录 http://222.18.159.122:8010 页面来查看集群的整体情况。

7.5 数据的加载、切分和分发

HPCC 是一种面向大数据的高性能计算集群，大数据处理最为简单的方法就是将数据分布到集群中大量的节点上进行分布式并行处理，因此数据的加载、切分和分发是进行数据处理的基础。HPCC 的分布式文件系统是与系统紧密结合的，在进行数据处理和分析，系统可以通过文件的逻辑目录顺利地找到被分割后的文件块并对该文件块进行处理。HPCC 的这一做法与 Hadoop 很相似，Hadoop 中的 HDFS 文件系统也正是在完成这一任务。数据的切分是实现计算向数据迁移的重要步骤。

数据的加载、切分和分发可以利用 HPCC 提供的 ECLWatch 来完成，ECLWatch 是一个网页化的管理界面，连接方式是 Thor 主节点 IP 地址（ESP 服务器）+8010 端口。

（1）数据的加载：上传数据到 HPCC 系统。

在 HPCC 系统中，原始数据（没有被切分和分发数据）的存放位置位于在 Thor 集群的 Master 节点（本节中 Master 节点为 node1）的 /var/lib/HPCCSystems/mydropzone 目录。对于 2GB 以下的小文件，可以利用 ECLWatch 来完成文件的上传，对于大文件则应采用 WinSCP 等第三方安全传输工具将数据上传到对应的目录。这里我们只介绍利用 ECLWatch 上传数据文件。

利用浏览器登录 HPCC 系统 ESP 服务器的 8010 端口，即可以看到 ECLWatch 的主界面，如图 7.8 所示。

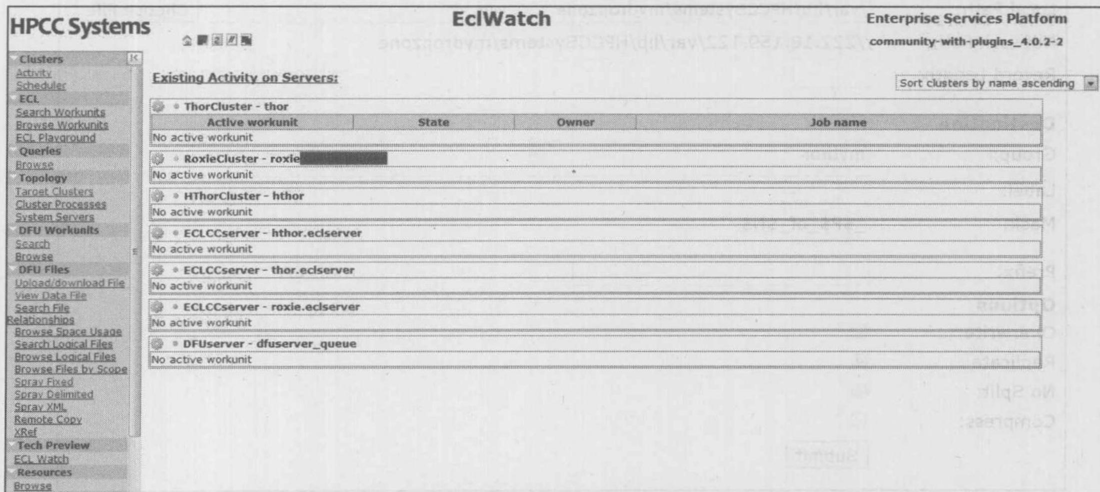


图 7.8 ECLWatch 主页面

单击 ECLWatch 页面上的“DFU Files”→“Upload/download File”，单击“选择文件”按钮，选择需要上传的文件，单击“Upload Now”完成文件向 Thor 集群主节点的上传工作。文件上传后保存在 Thor 集群主节点的/var/lib/HPCCSystems/mydropzone 目录下。我们也可以使用 FTP 工具或其他任何方式将数据复制到 HPCC 主节点的这一目录，效果与 ECLWatch 工具上传的数据一样。文件存放在 Thor 集群主节点中是为下一步对文件进行切分和分发做准备。通过这一页面我们也可以将已保存在主节点上的文件下载和删除。这一步我们将一个测试数据 OriginalPerson 上传到系统。

（2）对源数据进行切分，分发到各个节点。

数据上传到系统后为了对海量数据实现分布式的并行处理，HPCC 系统在进行数据处理之前，会首先对选定的数据文件进行切分，并将切分后的文件分发到各个 Slave 节点，供 Thor 集群进行处理。对文件的切分既可以采取固定长度切分（Spray Fixed），也可以根据用户自己定义的如逗号、分号等来进行切分（Spray Delimited）。HPCC 系统的 ECLWatch 提供了 Web 接口的界面化数据切分页面，我们可以通过设置数据切分参数，采取设定的方式对数据进行切分。

这里我们采用固定长度的切分方法。单击“DFU Files”→“Spray Fixed”，可以看到如图 7.9 所示的页面。

Spray Fixed

Source

Machine/dropzone: node159122/mydropzone
IP Address: 222.18.159.122
Local Path: /var/lib/HPCCSystems/mydropzone [Choose File]
Network Path: //222.18.159.122/var/lib/HPCCSystems/mydropzone
Record Length:

Destination

Group: mythor
Label:
Mask: ._P_of_N
Prefix:

Options

Overwrite:
Replicate: ✔
No Split:
Compress:

[Submit]

图 7.9　文件切分发布页面

“Machine/dropzone”为加载文件所在的节点和区域；

“IP Address”为 Thor 集群主节点的 IP 地址；

“LocalPath”为将要进行切分的数据所在位置，单击“Choose File”按钮，这里选择刚刚上传到主节点 /var/lib/HPCCSystems/mydropzone 目录下的数据文件 OriginalPerson 数据，OriginalPerson 数据集大小为 100MB 左右，总共有 841400 条记录，每条记录长度为 124 字节，记录中的每个字段长度小于定义长度时用空格补足，每条记录的描述如下：

表 7.2 OriginalPerson 实验数据描述

字段名	类型	描述
FirstName	15 Character String	First Name
LastName	25 Character String	Last name
MiddleName	15 Character String	Middle Name
Zip	5 Character String	ZIP Code
Street	42 Character String	Street Address
City	20 Character String	City
State	2 Character String	State

“Record Length”为切分数据每条记录的长度，OriginalPerson 数据的每条记录为 124 字节，所以在“Record Length”对应的编辑框中填入 124，这样系统在分割文件时可以保证不会将一条记录放在不同的节点上。

在“Label”键入“test::OriginalPerson”，这一标志对应今后此数据切分后的逻辑位置，单击“Submit”按钮，即可看到如图 7.10 所示的界面，单击“ViewProgress”按钮可以查看数据切分和分发进度情况。

DFU Workunit Details

Field	:	Value
ID	:	D20131230-160218
ClusterName	:	thor
JobName	:	originalperson
DFUServerName	:	mydfuserver
Queue	:	dfuserver_queue
Protected	:	
Command	:	Spray
TimeStarted	:	2013-12-30 08:02:18
TimeStopped	:	2013-12-30 08:02:31
PercentDone	:	View Progress
ProgressMessage	:	100% Done, 0 secs left (104/104MB @27355KB/sec) current rate=27355KB/sec [4/4nodes]
SummaryMessage	:	Total time taken 13 secs, Average transfer 27355KB/sec
State	:	finished
SourceIP	:	222.18.159.122
SourceFilePath	:	/var/lib/HPCCSystems/mydropzone/OriginalPerson
SourceRecordSize	:	124
SourceFormat	:	FIXED
SourceNumParts	:	1
SourceDirectory	:	/var/lib/HPCCSystems/mydropzone
DestLogicalName	:	test::originalperson
DestGroupName	:	mythor
DestDirectory	:	/var/lib/HPCCSystems/hpcc-data/thor/test/
DestNumParts	:	4
MonitorSub	:	0
Overwrite	:	0
Replicate	:	1
Compress	:	0
AutoRefresh	:	0

Save Reset

图 7.10　DFU 详细信息

进入 Thor 集群中的子节点的目录/var/lib/HPCCSystems/hpcc-data/thor/test/下，我们能看见类似这样的文件 originalperson._1_of_4，该文件就是被系统切分后分发到子节点上的文件，由于我们的实验平台有 4 个子节点，所以文件被切分为了 4 个块，这是其中的第 1 个块，用户在数据处理时并不需要直接操作这些切分后的块文件，只需要对文件的逻辑位置进行操作就行了，系统会自动实现分布式的数据处理。同时根据备份策略，系统在/var/lib/HPCCSystems/hpcc-mirror/thor/test/目录下保存一个存储于其他节点的数据块，如 originalperson._4_of_4，这种交叉备份可以在有一个节点失效时数据能被恢复。因此 HPCC 中的数据切分和分发过程就是将数据向集群文件系统的分发过程，这个过程完成后数据就是被分布式地存储于 HPCC 集群中的，这就为下面对数据进行分布式分析提供了可能。

7.6　ECL 语言基础知识

ECL（Enterprise Control Language）是 HPCC 系统主要的编程语言。ECL 将其代码编译为优化的 C++代码，然后编译为 Thor 和 Roxie 平台上可运行的 DLL 库文件来执行，从而获得了高速性能。ECL 非常灵活，任何外部语言如 C++、Java 都可以在外部编写，然后编译为 ECL 可以调用的函数库。

ECL 是 HPCC 平台与其他数据密集型计算解决方案的主要区别之一。使用 ECL 语言无需开发人员关注集群的计算节点数目，大大降低了并行编程难度，提高应用开发人员的效率。ECL 语言专门用于数据操作和数据查询，可以对以往无法处理的海量数据进行操作。

ECL 是一种优化的以数据为中心的说明性程序语言（declarative language），但它继承了很多函数语言的特点。ECL 语言主要擅长应用于对数据的分析处理和查询等工作。ECL 提供 TRANSFORM、JOIN、PROJECT、SORT、DISTRIBUTE、MAP 等高级数据操作，大大降低了项目的开发难度。ECL 语言隐藏了数据处理时的并行处理过程，在程序设计时不用考虑系统是如何完成数据并行处理的。ECL 没有采用 Hadoop 系统中 Key-Value 这种数据结构，能实现对数据的复杂查询、多键值查询和模糊匹配能功能。由于 HPCC 系统是面向大数据的系统，大量数据的移动代价是很大的，ECL 语言尽可能地减少数据的移动，采用将程序向数据移动的策略，而不是将数据向程序移动，比较而言程序的大小会远远小于数据的大小，这种移动策略也是大多数大数据系统的策略。ECL 语言看上去像是运行在分布式环境下的数据分析语言。

从功能上讲 ECL 程序包含两种代码：定义和需要执行的操作。ECL 语言的语法和操作很少，学习起来较为方便。大多数 ECL 代码都是由定义组成，同时 ECL 语言是大小写无关。

7.6.1 ECL 语言的保留关键字

本章的实例部分会使用一些 ECL 语言的保留关键字，下面对这些关键字的用法进行简单介绍。

1. EXPORT 关键字

EXPORT 关键字用于指定能够被其他定义所引用的定义，EXPORT 关键词后跟着定义名称和具体内容。如下代码定义 MyDef 为字符串'hello'，保存这段代码的文件为 MyDef.ecl，存放于 test 文件夹下，注意这里的文件名需要和定义的名称相同。这样在其他代码文件中可以利用 test.MyDef 来引用这个定义。

```
EXPORT MyDef:= 'hello';
```

2. IMPORT 关键字

IMPORT 关键字使 EXPORT 所指定的定义在该代码文件中可用。如下代码在 test 文件夹下新建的代码文件中利用 IMPORT 关键词引入 test 文件夹下的定义，由于 MyDef 在 test 文件夹下，所以可以在这段代码中引用 MyDef.ecl 文件中对 MyDef 的定义，这段代码的输出为'helloworld!'

```
IMPORT test;
Hello:=test.MyDef+'world!';
OUTPUT(Hello);
```

3. SELF 关键字

SELF 关键字用于在 TRANSFORM 中作为记录某一字段变化后值的存储。

4．GROUP 关键字

GROUP 关键字主要用于输出和 TABLE 定义中的分类聚集操作，与 SQL 的 GROUP BY 相似。以下代码实现将示例数据按邮编 Zip 进行分类计数输出为交叉表。

```
IMPORT Test;
/*以交叉表方式输出*/
wordCountTable :=
TABLE(Test.File_OriginalPerson,{Test.File_OriginalPerson.Zip,COUNT(GROUP)},
Zip);
OUTPUT(wordCountTable);
```

7.6.2 ECL 语言的记录定义和操作

ECL 是一种面向记录的语言，对记录的操作是 ECL 语言的一个重要部分，体现了 ECL 语言在记录操作上的优势。

1．记录定义声明 RECORD

RECORD 声明用于定义记录的结构，以 END 结束。一个典型的 RECORD 定义如下：

```
R := RECORD
STRING10 fname;
STRING12 lname;
END;
```

2．记录集定义声明 DATASET

DATASET 声明用于定义在内存和文件中生成记录集，记录的结构由 RECORD 声明所定义。以下代码根据示例数据生成记录集，记录集的名称为 File_OriginalPerson。DATASET 中的第一个参数 '~Test::OriginalPerson' 指定生成记录集的原始数据的逻辑位置，Test.Layout_People 指定记录的结构，THOR 参数指定文件类型为 THOR。生成的记录集保存在 File_OriginalPerson 中。

```
IMPORT Test;
File_OriginalPerson :=
DATASET('~Test::OriginalPerson',Test.Layout_People,THOR);
```

3．创建索引 BUILD

BUILD 的功能是创建一个索引文件，以下代码为记录集 File_OriginalPerson 创建一个由 Zip 作为索引的索引文件，并将索引文件存放在逻辑位置 test::PeopleByZipINDEX，其中 fpos 为每条记录的在文件中位置标识。

```
BUILD(File_OriginalPerson,{Zip,fpos},'~test::PeopleByZipINDEX');
```

4．索引引用 INDEX

INDEX 引用一个已被创建的索引文件，INDEX 与 BUILD 创建的索引是对应的，如果我

们已在系统中创建了索引文件 test::PeopleByZipINDEX，则索引的引用代码如下：

```
MyIndex:=INDEX(File_OriginalPerson,{zip,fpos},'~test::PeopleByZipINDEX');
```

MyIndex 为索引记录集的名称，INDEX 中的第一个参数是原始记录集，第二个参数为索引项，第三个参数为索引文件的逻辑位置。

5．记录变换 TRANSFORM

TRANSFORM 实现记录的变换，它必须作用于整个记录集并且需要以 END 关键字结束。

7.6.3 ECL 语言集成开发环境

HPCC 提供一个进行 ECL 语言开发的集成开发环境——ECL IDE，ECL IDE 是运行在 Windows 下的客户端应用程序，通过 ECL IDE 我们可以实现针对 HPCC 系统的远程程序设计和运行。ECL IDE 的安装非常简单，运行安装程序即可。

使用启动 ECL IDE 时需要填写 HPCC 主服务器上的用户名和密码，如图 7.11 所示。

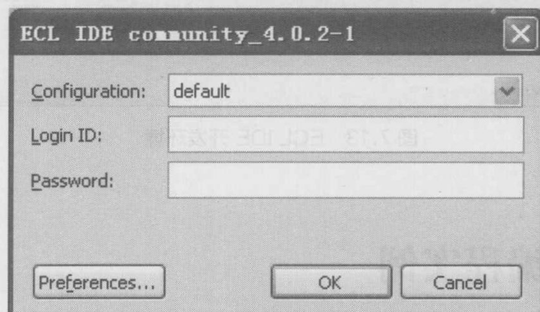


图 7.11 ECL IDE 登录界面

如果是第一次使用，首先需要单击“Preferences”按钮，配置 HPCC 中 Thor 主服务器的 IP 地址。在 Server 编辑框中填入 Thor 集群主服务器的 IP 地址，如图 7.12 所示。

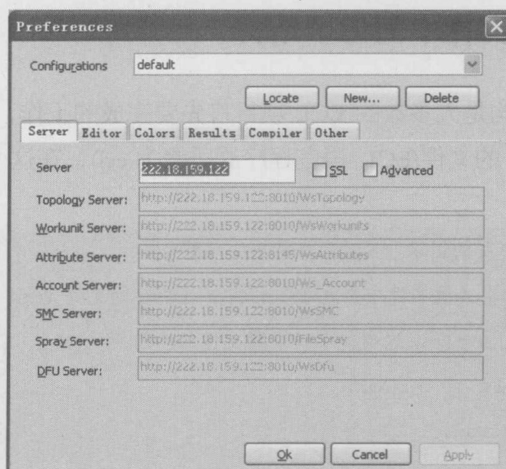


图 7.12 ECL IDE 登录配置界面

登录进入 ECL IDE 后的界面如图 7.13 所示，在这个环境中我们就可以开始 ECL 程序的设计了。ECL 集成开发环境中最主要的部分就是程序编辑框，ECL 程序的编辑和修改就是在这里完成的，同时程序编辑框也作为计算结果的显示和处理框使用，同时集开发环境中经常用到的部分还有语法错误提示框（syntax error）、错误日志框（error log）、程序存储框（repository）。

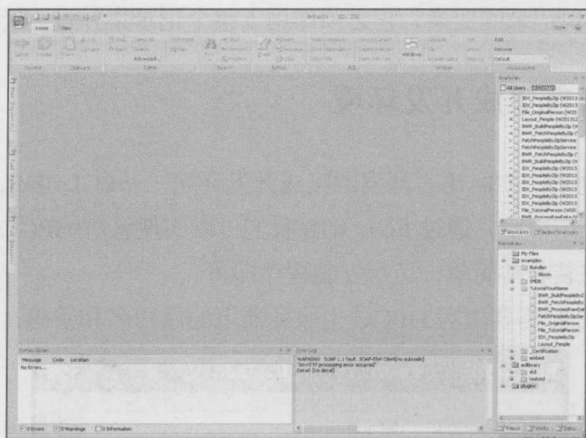


图 7.13　ECL IDE 开发环境

7.7　ECL 语言编程实例

为方便介绍，以下实例都是在 MyFiles 下的 Test 文件夹中完成的，所以首先在 My Files 下建立一个名称为 Test 的文件夹作为存放 ECL 代码的文件夹。下面实例中的数据就是前面上传的 OriginalPerson 数据。

7.7.1　声明数据文件中的记录结构

定义文件中的记录结构是大多数的 ECL 项目首先要完成的工作，在 Test 文件夹下添加一个文件名为 Layout_People 的文件(ECL 语言程序的后缀为.ecl)，在这个文件中定义每一条记录中各个字段的数据类型。

```
/*Layout_People 是声明的记录(RECORD)结构的名称*/
/*EXPORT 关键词表明如果其他文件中引入了 IMPORT Test 则允许 Test 文件夹中其他文件中的定义引用此定义，Test.Layout_People，这类似于一个跨文件的全局声明*/
EXPORT Layout_People := RECORD
    STRING15 FirstName;
    STRING25 LastName;
    STRING15 MiddleName;
    STRING5 Zip;
```

```
    STRING42 Street;
    STRING20 City;
    STRING2 State;
END;
```

在这段代码中我们定义了每一个记录中的字段名和字段长度，以 END 结束定义。由于这一定义被声明为 EXPORT，所以在后面的代码中可以通过，Test.Layout_People 来引用这一记录的定义。要注意的是这里定义名要与文件名相同，都是 Layout_People。Layout_People 记录的这一定义与之前上传的示例数据的存储结构是一致的，且长度都是 124 字节。

7.7.2 读取数据文件生成数据集

在 Test 文件夹下建立新文件，文件名为 File_OriginalPerson。

```
IMPORT Test;
EXPORT File_OriginalPerson :=
DATASET('~Test::OriginalPerson',Test.Layout_People,THOR);
```

这段代码生成原始数据的数据集，由于该数据集被声明为 EXPORT 该数据集名称 File_OriginalPerson 可以被其他文件引用。其中 DATASET 声明一个数据文件中的记录集，而每一条记录的结构则由 RECORD 进行声明。DATASET 的第一个参数为数据文件的逻辑位置，第二个参数为数据文件中的记录结构，第三个参数为文件类型。

7.7.3 统计记录条数

在 Test 文件夹下添加一个文件名为 File_OriginalPersonCount 的文件，输入以下代码。

```
/*IMPORT关键词使我们在前面声明的EXPORT声明记录Layout_people和File_OriginalPerson
在这段 ECL 代码中可以引用*/#
IMPORT Test;
COUNT(Test.File_OriginalPerson);
```

COUNT 函数对数据集中的记录条数进行计算。

按“F7”键可以检查代码的语法，如果语法没有错误，单击“Submit”按钮提交服务器执行。以上代码的功能是统计数据记录的总统条数，执行的结果如图 7.14 所示，结果显示，原始数据中共有 841400 条记录。

##	Res...
1	841400

图 7.14 程序运行结果

在提交运行统计记录条数这个程序后在 Thor 集群的每个子节点的 ~HPCCSystems/queries/mythor_20100/文件夹下生成了一个与这次程序提交时的时间戳名称相同的.so 动态连接库文件，如图 7.15 中的文件 V349257874_libW20140105-134510.so。从 HPCC 的系统架构来看数据文件是被切割后分布式地存储于各个节点上的，COUNT 操作需要读取完整的数据，所以系统将程序分别注入各个节点进行执行，在所有子节点上都能看到 V349257874_libW20140105-134510.so 这个动态连接库文件，该文件就是被注入的动态连接库程序，各节点分别将自己节点的文件块中的记录进行计数，汇总后即是该程序的输出，这一过程对节点而言是并行化的。这种并行化方法是基于数据切分的并行化方法，这一过程与 Hadoop 中的 MapReduce 过程非常相似。HPCC 这种处理方法向使用者屏蔽了并行数据处理的复杂性，从程序代码上完全看不出并行化的特征，看上去就像是对单个逻辑文件在进行操作，系统自动通过程序注入的方式实现了对数据处理的并行化。这种基于数据切分的方法是大数据系统并行计算较为通用的实现方法，具有内嵌的分布式文件系统是实现这一方法的基础，Hadoop 的 HDFS 分布式文件系统和 Google 的 GFS 分布式文件系统都起到了这个作用。

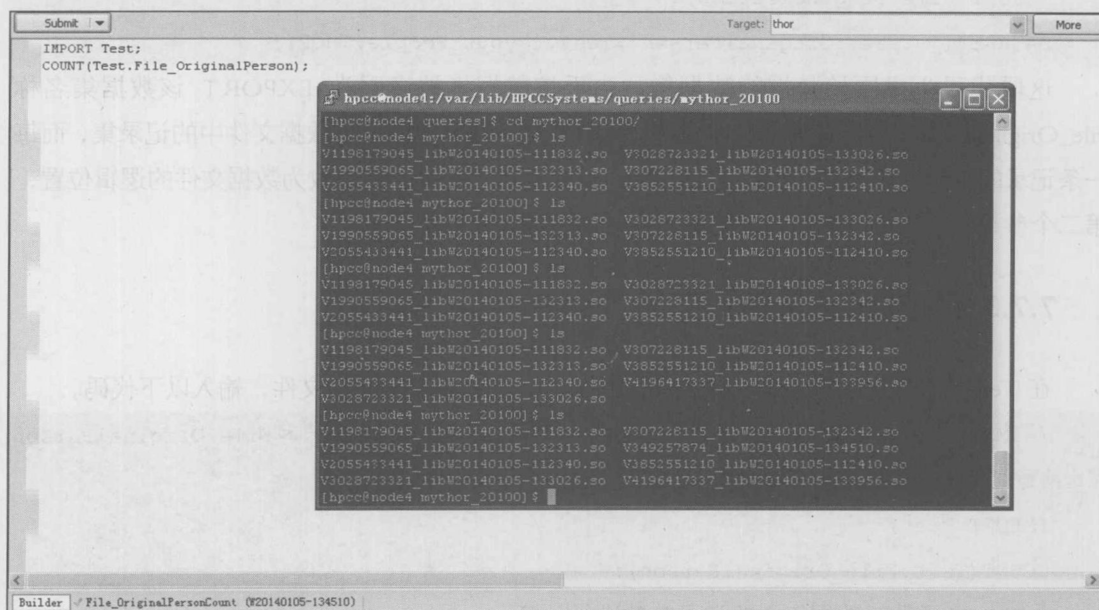


图 7.15　数据分布式存储示例

7.7.4　将数据集中的小写字母改为大写

将数据集中的所有记录姓名部分全部改为大写，这种对记录集的操作代表了一大类的记录集操作，可以利用 TRANSORM 来完成这一处理。

```
IMPORT Test,Std;
Test.Layout_People toUpperPlease(Test.Layout_People pInput):= TRANSFORM
```

```
    SELF.FirstName := Std.Str.ToUpperCase(pInput.FirstName);
    SELF.LastName := Std.Str.ToUpperCase(pInput.LastName);
    SELF.MiddleName := Std.Str.ToUpperCase(pInput.MiddleName);
    SELF.Zip := pInput.Zip;
    SELF.Street := pInput.Street;
    SELF.City := pInput.City;
    SELF.State := pInput.State;
END ;
OrigDataset := Test.File_OriginalPerson;
UpperedDataset := PROJECT(OrigDataset,toUpperPlease(LEFT));
OUTPUT(UpperedDataset,,'~test::TestPerson',OVERWRITE);
```

在函数 toUpperPlease 前定义输出的数据结构为 Test.Layout_People 中定义的结构，TRANSFORM的定义调用Std.Str.ToUpperCase()函数实现姓名所涉及的三个字段由小写改为大写，其他字段大小写不变。SELF 指定记录中各字段的输出。程序运行结果如图 7.16 所示，所有姓名都变为了大写。

##	firstname	lastname	middlename	zip	street	city	state
1	CHERIANNE	KHATCHATOURIAN	N	54530	69 BOULDER RIDGE RD # 25A	HAWKINS	WI
2	MUYESSER	RAPLEE	X	20747	55 SWAMP RD	DISTRICT HEIGHT	MD
3	ROSELIN	VICECONTE		97828	107 HILL TER	ENTERPRISE	OR
4	INDA	PROVINES		72941	290 W MOUNT PLEASANT AVE	LAVACA	AR
5	INDERDEEP	LAURENCE	D	32330	44 PROSPECT PL	GREENSBORO	FL
6	CHRYSTINE	MANGIAPANE		80007	1806 1ST AVE APT 8F	ARVADA	CO
7	ADELENE	STOCK	R	19901	1117 FARM RD	DOVER	DE
8	MENDY	RUFENBLANCHETTE		29697	3 W 83RD ST APT 4C	WILLIAMSTON	SC
9	LANNIE	AMERANTES	I	25312	200 W 20TH ST APT 909	CHARLESTON	WV
10	TARE	GONYEAU	T	79924	6 CANDLE CT	EL PASO	TX
11	FINNEY	ARISTILDE	P	31220	222 1ST AVE APT 2B	MACON	GA

图 7.16　大小写转换程序运行结果

大小写转换后的文件被重命名为 testperson，存放于逻辑位置 ~ test/testperson 中，而文件的物理存储方式仍然是分布式的，testperson 文件被分为了 4 块存放于 Thor 集群的 4 个子节点上，因此我们在 Thor 集群的子节点的目录/var/lib/HPCCSystems/hpcc-data/thor/test/下也能发现 testperson._1_of_4 这样被切分后的文件。

7.7.5　建立索引实现对数据集的检索

要实现记录数据集的高速查询就需要建立索引，索引是对数据库表中一个或多个列的值进行排序的结构。在下面的例子中我们利用邮编(Zip)作为索引项，为了根据索引定位记录位置，建立索引时在每个记录后面增加了一个记录位置项，记录位置表明每条记录在文件中对应的物理位置。

（1）建立索引文件（MyZipIndex.ecl）。

```
IMPORT Test;
File_OriginalPerson :=
   DATASET('~test::OriginalPerson',{Test.Layout_People,UNSIGNED8 fpos
{virtual(fileposition)}},THOR);
/*以记录中的邮编（Zip）作为索引项，索引文件名为 PeopleByZipINDEX*/
/*建立索引文件，并写入 test/PeopleByZipINDEX 文件*/
BUILD(File_OriginalPerson,{Zip,fpos},'~test::PeopleByZipINDEX');
MyIndex:=INDEX(File_OriginalPerson,{zip,fpos},'~test::PeopleByZipINDEX');
/*输出索引数据集*/
OUTPUT(MyIndex);
```

生成数据集时利用 UNSIGNED8 fpos {virtual(fileposition)}在每条记录中增加了一个字段 fpos，该字段标明记录在文件中的位置。这样在利用 Zip 进行索引时可以通过 fpos 字段直接找到记录在文件中的位置。

BUILD 实现索引的建立并将索引文件写入逻辑位置 test/PeopleByZipINDEX 文件，索引文件不需要重复建立，INDEX 生成索引数据集，图 7.17 为生成的索引数据集的情况。

在子节点的目录/var/lib/HPCCSystems/hpcc-data/thor/test/下可以发现文件 peoplebyzipindex._1_of_5，这证明 HPCC 系统的索引文件也是在子节点上分布式存储的，因此系统基于索引的检索工作也是分布式检索的。

##	zip	fpos
1	00603	2233364
2	00603	3141540
3	00603	5515520
4	00603	6853976
5	00603	8632880
6	00603	13807276
7	00603	15210584
8	00603	15941812
9	00603	16415368
10	00603	17324660
11	00603	19950732

图 7.17　索引数据集

（2）利用索引进行检索（SearchZipIndex.ecl）。

索引建立后就可以依据索引对数据集进行检索操作，代码如下：

```
IMPORT Test;
File_OriginalPerson :=
  DATASET('~test::OriginalPerson',{Test.Layout_People,UNSIGNED8 fpos
{virtual(fileposition)}},THOR);
MyIndex:=INDEX(File_OriginalPerson,{zip,fpos},'~test::PeopleByZipINDEX');
/*利用索引检索邮编为 33024 的所有记录*/
ZipFilter :='33024';
FetchPeopleByZip :=
  FETCH(File_OriginalPerson,MyIndex(Zip=ZipFilter),RIGHT.fpos);
OUTPUT(FetchPeopleByZip);
```

示例程序利用索引从数据集中检索出 Zip=33024 的所有记录，检索结果如图 7.18 所示。

##	firstname	lastname	middlename	zip	street	city	state	fpos
1	Zhiagn	Archie	W	33024	3 INGRID RD	PEMBROKE PINES	FL	197284
2	Ashanti	Tsames		33024	200 HOWE RD	PEMBROKE PINES	FL	755408
3	Kirakos	Robshaw		33024	1630 AUGUSTA DR	PEMBROKE PINES	FL	2663520
4	Leesha	Montalvo		33024	1 ASTOR PL APT 5N	PEMBROKE PINES	FL	5040724
5	Reynold	Bellero		33024	489 WOLCOTT ST APT 75	PEMBROKE PINES	FL	7349852
6	Karolyn	Scatena		33024	2 NEW PROVIDENCE AVE	PEMBROKE PINES	FL	8961728
7	Authea	Vardi	W	33024	889 BROADWAY APT 7A	PEMBROKE PINES	FL	9249780
8	Eteuini	Scharer	K	33024	214 SPRUCE HILLS DR	PEMBROKE PINES	FL	10036436
9	Samuel	Trovati		33024	7 DEER RUN DR	PEMBROKE PINES	FL	12616380
10	Tasey	Zitniak	J	33024	235 W 76TH ST APT 2E	PEMBROKE PINES	FL	13228072
11	Lavance	Gippert		33024	55 MOOSE HILL RD	PEMBROKE PINES	FL	13735108

图 7.18　检索结果

7.7.6　发布数据

通过 Web 访问已建立索引的数据，需要将数据进行发布，在 test 文件夹下加入文件名为 MyZipPublish 的新文件，写入如下代码：

```
IMPORT Test;
File_OriginalPerson :=
  DATASET('~test::OriginalPerson',{Test.Layout_People,UNSIGNED8 fpos
{virtual(fileposition)}},THOR);
MyIndex:=INDEX(File_OriginalPerson,{zip,fpos},'~test::PeopleByZipINDEX');
STRING10 ZipFilter := '' :STORED('ZIPValue');
resultSet :=
```

```
FETCH(File_OriginalPerson,MyIndex(Zip=ZipFilter),RIGHT.fpos);
OUTPUT(resultset);
```

选择目标集群为 thor，但单击“Submit”按钮提交，在 Thor 集群发布数据，如图 7.19 所示。

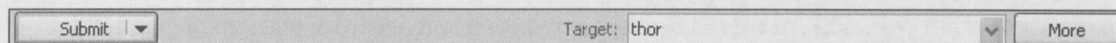


图 7.19　Thor 集群发布

运行成功后单击图 7.20 中的“ECL Watch”按钮。

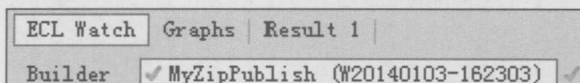


图 7.20　“ECL Watch”按钮

可以看到如图 7.21 所示界面：

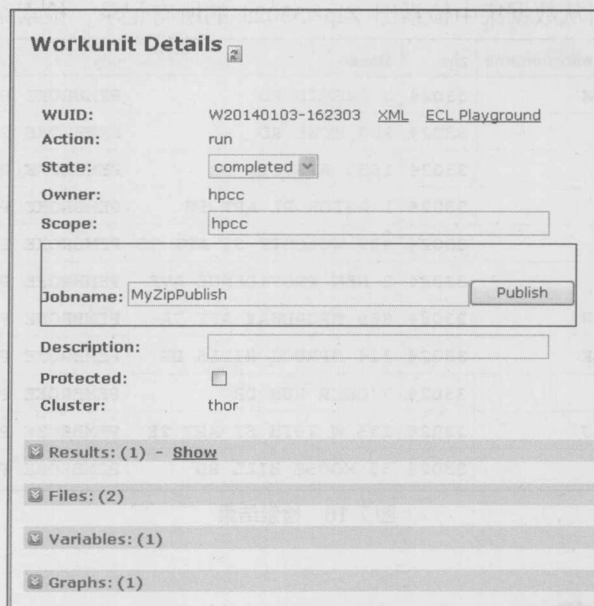


图 7.21　Thor 集群发布界面

正常的话可以看到状态项（State）对应的为“completed”，集群（Cluster）为 thor 集群，Jobname 处系统自动填写为文件名 MyZipPublish。单击发布按钮“Publish”完成发布。

发布成功后利用浏览器访问主节点 IP:8002 端口，进入 WsECL Web service 的主页面，展开页面上的 Thor 树状控件，选择 MyZipPublish，可以看到图 7.22 所示的界面。

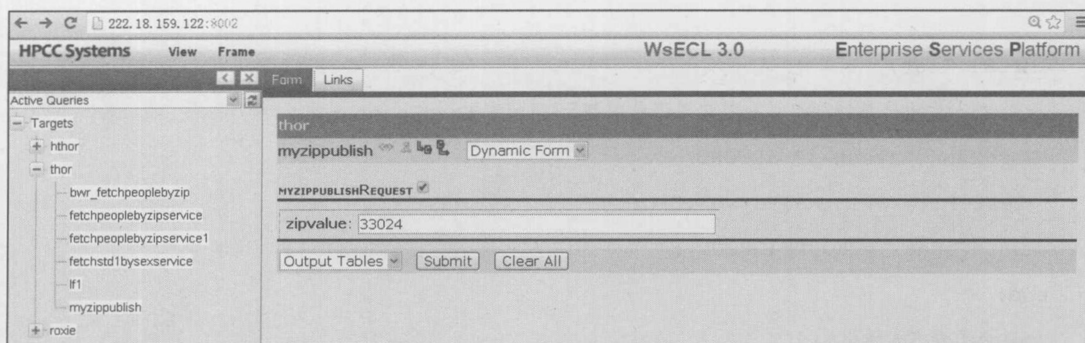


图 7.22　WsECL Web service 的主页面

在搜索框中填入需要搜索的邮编值“33024”，单击“Submit”按钮，即可得到数据集中所有邮编值为“33024”的记录列表，如图 7.23 所示。

myzippublish Response

Dataset: Result 1

	firstname	lastname	middlename	zip	street	city	state	fpos
1	Zhiagn	Archie	W	33024	3 INGRID RD	PEMBROKE PINES	FL	197284
2	Ashanti	Tsames		33024	200 HOWE RD	PEMBROKE PINES	FL	755408
3	Kirakos	Robshaw		33024	1630 AUGUSTA DR	PEMBROKE PINES	FL	2663520
4	Leesha	Montalvo		33024	1 ASTOR PL APT 5N	PEMBROKE PINES	FL	5040724
5	Reynold	Bellero		33024	489 WOLCOTT ST APT 75	PEMBROKE PINES	FL	7349852
6	Karolyn	Scatena		33024	2 NEW PROVIDENCE AVE	PEMBROKE PINES	FL	8961728
7	Authea	Vardi	W	33024	889 BROADWAY APT 7A	PEMBROKE PINES	FL	9249780
8	Eteuini	Scharer	K	33024	214 SPRUCE HILLS DR	PEMBROKE PINES	FL	10036436
9	Samuel	Trovati		33024	7 DEER RUN DR	PEMBROKE PINES	FL	12616380
10	Tasey	Zitniak	J	33024	235 W 76TH ST APT 2E	PEMBROKE PINES	FL	13228072

图 7.23　Thor 集群检索结果

利用 Roxie 发布时，Target 处选择“Roxie”，提交时选择“Compile”，其他操作过程与 Thor 集群发布相似。

7.7.7　HPCC 中的 WordCount 操作

WordCount 操作是大数据系统中常见的操作，在 Hadoop 中也常常作为典型的案例来进行讲解，HPCC 系统同样能非常简单地实现对海量数据的 WordCount 操作。本节将以 WordCount 程序为例介绍 HPCC 程序的运行机制。

WordCount 就是对文件中的单词出现的次数进行统计，在 Hadoop 系统中可以通过一个典型的 MapReduce 过程来实现这一功能，而 HPCC 提供了功能更强大、算法更简单的实现方法。

HPCC 是面向记录的，在下面的案例中我们对 Zip 字段做 WordCount 操作，统计每个 Zip 值在记录中出现的次数。

```
IMPORT Test;
/*以 Zip 作为计数的分组，从而实现记录中基于 Zip 的 WordCount 操作*/
 WordCountLayout := RECORD
  Test.File_OriginalPerson.Zip;
  wordCount := COUNT(GROUP);
end;
/*以交叉表方式输出*/
wordCountTable := TABLE(Test.File_OriginalPerson, WordCountLayout, Zip);
OUTPUT(wordCountTable);
```

对 Zip 字段进行 WordCount 操作后的结果如图 7.24 所示。#

##	zip	wordcount
1	00603	46
2	00669	39
3	00682	27
4	00683	42
5	00692	37
6	00698	39
7	00716	36
8	00726	34
9	00729	36
10	00735	40
11	00742	58
12	00766	42
13	00801	33
14	00804	39
15	00820	43

图 7.24　Zip 的 WordCount 结果

与此类似，也可以对文字内容进行 WordCount 操作，代码如下：

```
WordLayout := RECORD
  STRING word;
end;
wordsDS := DATASET([{'HPCC'}, {'HPCC'}, {'BIGDATA'}, {'FAST'}, {'FAST'},
{'CLUSTER'},
          {'PARALLEL'}, {'BI'}, {'ANALYTICS'}, {'ML'} ], WordLayout);
```

```
WordCountLayout := record
  wordsDS.word;
  WordCount := COUNT(GROUP);
end;
wordCountTable := TABLE(wordsDS, WordCountLayout, word);
OUTPUT(wordCountTable);
```

输出结果如图 7.26 所示。

1	ANALYTICS	1
2	BI	1
3	BIGDATA	1
4	CLUSTER	1
5	FAST	2
6	HPCC	2
7	ML	1
8	PARALLEL	1

图 7.25　文字的 WordCount 结果

限于篇幅，本书未对 ECL 语言进行详细介绍，只是通过一些简单例子使大家能对 ECL 的主要功能和编程方法有所了解。ECL 语言是与 Thor 集群、Roixe 集群紧密结合的语言，通过与 HPCC 的分布式存储系统相结合实现了面向数据自动并行计算能力。ECL 语言里基本没有明显的并行化痕迹，编程人员无需了解集群的具体结构以及程序并行化执行的任何细节，程序从表明上看与串行程序完全一样，这大大降低了并行化编程的难度，是 ECL 语言的重要优点之一。ECL 语言自动并行化的实现离不开与之结合的分布式文件系统支持，正是有了这个文件系统的支持 ECL 才能实现自动并行化，分布式文件系统为用户隔离了数据分布式存储复杂细节，这一点在多数大数据系统都能体现出来。

练习题

1. 简述 HPCC 的主要特点。
2. 高性能计算目前可以分为两类：一类是面向＿＿＿＿＿＿＿的高性能计算；另一类是面向＿＿＿＿＿＿＿＿＿的高性能计算。
3. 数据密集型集群计算系统主要有＿＿＿＿＿＿、＿＿＿＿＿＿、＿＿＿＿＿＿、＿＿＿＿＿＿、＿＿＿＿＿＿、＿＿＿＿＿＿。

4. HPCC 系统从总体物理上可以看作在同一个集群上部署了__________、__________、两套集群计算系统。

5. HPCC 的系统服务器包含__________、__________、__________、__________、__________。

6. 简述 HPCC 平台数据检索任务的执行过程。

7. 熟悉 HPCC 网页化管理界面 ECLWatch。

8. 动手完成 HPCC 的安装部署。

PART 8

第 8 章 Storm——基于拓扑的流数据实时计算系统

大数据处理有批处理和流处理两种模式。批处理模式下，数据源是静态的，使用这种处理模式的系统有 Hadoop、Disco、Spark 等；流处理模式下，数据源是动态的，使用这种处理模式的系统有 Storm、S4 等。通常地，批处理模式产生的中间结果会写入磁盘，而流处理模式产生的中间结果全部存入内存，读写磁盘会大大增加处理的延迟和处理的繁琐性，因此流处理模式相较于批处理模式，它的处理延迟更低，处理过程更加简单，更适合应用于实时计算。Storm 是一款典型的流处理模式下的大数据处理分析系统，与 Hadoop 等批处理系统相比，其在实时性、高效性、容错性、扩展性方面都表现出了明显的优势。

8.1 Storm 简介

BackType 公司（后被 Twitter 收购）前工程师 Nathan Marz，在使用 Hadoop 过程中，因为不满意 Hadoop 系统的扩展性和其代码的繁琐性，以及其粗糙的容错处理机制，提出了一种支持实时流处理、扩展机制简单的编程模型 Topology，取名为 Storm。Storm 于 2011 年 9 月 19 日正式开源，实现 Storm 的语言为一种运行于 Java 平台的 LISP 方言——Clojure。Storm 是很有潜力的流处理系统，出现不久，就在淘宝、百度、支付宝、Groupon、Facebook、Twitter 等平台上得到使用。第三方支付平台支付宝使用 Storm 来计算实时交易量、交易排行榜、用户注册量等，每天处理的信息超过 1 亿条，处理的日志文件超过 6TB；团购网站 Groupon 使用 Storm 对实时数据进行快速数据清洗、格式转换、数据分析；Twitter 使用它来处理 tweet（用户发送到 Twitter 上的信息）。

Storm 的 Topology 编程模型简单，在实际任务处理时却很实用。Topology，实际上就是任务的逻辑规划，包含 Spout 和 Bolt 两类组件，Spout 组件负责读取数据，Bolt 组件负责任务处理。与 MapReduce 相比，它的任务粒度相对灵活，不只局限于 Mapreduce 中的 Map()和 Reduce()函数，用户可以根据任务需求编写自己的函数。同时，它不存储中间数据，组件与组件之间的数据传递通过消息传递的方式，对于很多不需要存储中间数据的应用来说，Topology 编程模型降低了处理过程的繁琐与延迟。

（1）Storm 具有很好的容错性、扩展性、可靠性和健壮性。

Storm 使用 Zookeeper（Hadoop 中的一个正式子项目，后被广泛使用的一种分布式协调工

具）作为集群协调工具，当发现正在运行的 Topology 出错的时候，Zookeeper 就会告诉 Nimbus（Storm 系统的主进程，负责分发任务等操作），然后 Nimbus 就重新分配并启动任务。在 Storm 中，Topology 被提交后，在没有被手动杀死之前，它都将一直处于运行状态。这些措施都是为了保证该系统的容错性。Storm 采用三进程架构——Nimbus、Supervisor、Zookeeper，无论是集群还是单机都只有这三个进程。当需要在集群中新加入节点的时候，只需要修改配置文件和运行 Supervisor 和 Zookeeper 进程即可，扩展起来十分方便。另外，Storm 采用消息传递方式进行数据运算，数据传输的可靠性至关重要。Storm 系统中传递的消息，主节点都会根据消息的产生到结束生成一棵消息树。所以，消息从诞生到消亡的整个过程，它都会被跟踪。如果主节点发现某消息丢失，那么它就会重新处理该消息。正是因为有了容错性、可靠性的保障，该系统运行中体现出健壮性，不会出现轻易宕机、崩溃的现象。

（2）Storm 并行机制灵活。

各个组件的并行数由用户根据任务的繁重程度自行设定，如果该组件处理的任务复杂度高，耗费时间多，那么并行数目的设置就偏大些，相反地，并行数目的设置则偏小些。这样，拓扑中的每个组件就能很好地配合，最大化地利用集群性能，提高任务处理效率。

（3）Storm 支持多种语言。

Storm 内部实现语言是 Clojure，基于 Storm 开发的应用却可以使用几乎任何一种语言，而所需的只是连接到 Storm 的适配器。Storm 默认支持 Clojure、Java、Ruby 和 Python，并已经存在针对 Scala、JRuby、Perl 和 PHP 的适配器。更多的适配器将会随着应用的扩展变得更加的丰富。

8.2 Storm 原理及其体系结构

8.2.1 Storm 编程模型原理

Storm 编程模型采用的是生活中常见的并行处理任务方式——流水线作业方式。Storm 实现一个任务的完整拓扑如图 8.1 所示，在 Storm 中每实现一个任务，用户就需要构造一个这样的拓扑。该拓扑包含两类组件：Spout 和 Bolt。Spout 负责读取数据源，Bolt 负责任务处理。Storm 处理一个任务，往往会把该任务拆分为几部分，分别由不同的 Bolt 组件来实现。这是流水线作业中实现并行和提升任务处理效率采用的方法。

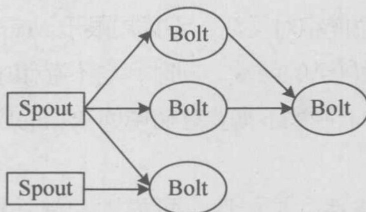


图 8.1 Storm 编程模型 Topology

比如，使用 Storm 处理单词统计的任务（WordCount），该任务的拓扑如图 8.2 所示。spout 组件负责读取要统计的数据源中的句子，split 组件负责将接收到的句子拆分成单个的单词，把这些单词发送至 count 组件，count 组件负责统计发送过来的单词出现的次数。



图 8.2 WordCount Topology

这样一个统计单词的任务就被拆分为三部分来操作，每部分可以根据任务的繁重程度来规划并行数目，各个组件的并行数没有明确规定。比如，可以设置 spout 并行数为 2，split 并行数为 8，count 并行数为 12，如图 8.3 所示。

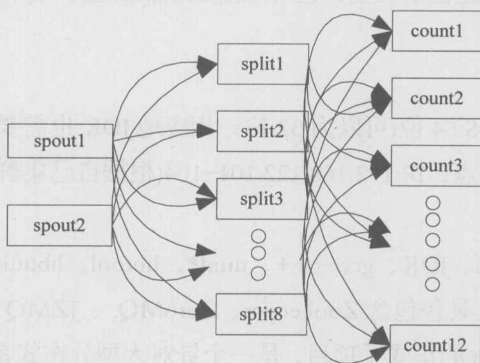


图 8.3 WordCount 并行工作模式

8.2.2 Storm 体系结构

Storm 中因为没有使用文件系统，相比于 Hadoop 它的架构要简单得多。Storm 依然采用的是主从架构模式，即有一个主进程和多个从进程。除了这两个进程以外，还有在主进程与从进程之间进行协调的进程 Zookeeper。Storm 的体系结构如图 8.4 所示。

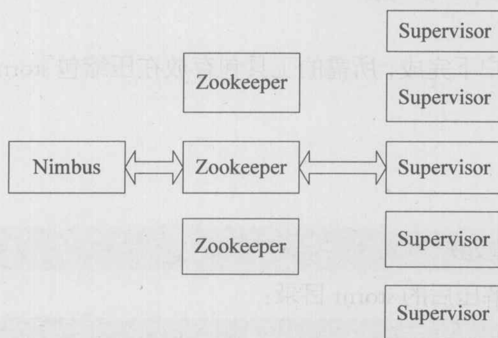


图 8.4 Storm 体系结构

知道了Storm是由三类进程组成，但是Storm的三进程部署到具体的集群上又是怎样的呢？因为主进程任务是负责分发任务和调度任务，在一个任务中只需要一个这种角色，所以主进程Nimbus只需要部署到一个节点上。而工作机进程是负责实际的任务处理，那么一个集群有多少节点配置多少个工作机进程，这样才能最大限度地利用集群性能，所以Supervisor需要部署到集群中的每一个节点上。Zookeeper进程负责主进程与工作进程协调的任务，因此它也需要部署到集群中的每一个节点上。知道了这点，下面的部署安装也就不难理解了。

8.3 搭建Storm开发环境

搭建Storm开发环境首先需要安装Storm系统需要的依赖包，然后再安装Storm系统工具包。Storm开发环境可以搭建在单机上，也可以搭建在集群上。本节我们在4个节点构建的集群上搭建Storm开发环境。

1．实验环境说明

（1）操作系统：CentOS 64位(可以为32位，相应地JDK也需要32位)。

（2）集群配置：4个节点，IP:192.168.122.101~104(根据自己集群情况自行设置）。

2．安装内容说明

（1）依赖软件：Python、JDK、gcc-c++、uuid*、libtool、libuuid、libuuid-devel。

（2）安装Storm所需工具包包含Zookeeper、ZeroMQ 、JZMQ 和Storm。

① Zookeeper：Hadoop的正式子项目，是一个针对大型分布式系统的可靠协调系统，提供配置维护、名字服务、分布式同步、组服务等功能。Zookeeper的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

② ZeroMQ：类似于Socket的一系列接口，ZeroMQ与Socket的区别在于Socket是端到端的（1:1）的关系，而ZeroMQ是N: M的关系，屏蔽细节使得网络编程更加简单。

③ JZMQ：针对ZeroMQ的Java binding。

④ Storm：Storm系统主程序，本文使用的Storm的版本号为0.8.1。

8.3.1 Storm的安装步骤

以下安装均在root用户下完成，所需的工具包存放在压缩包storm.tar.gz中，所以首先要把该压缩包解压。

（1）安装准备。

解压storm.tar.gz包：

```
tar -xzvf storm.tar.gz
```

切换当前工作目录到解压后的storm目录：

```
cd storm
```

（2）安装依赖文件。

使用 yum 方式安装依赖包 g++、uuid*、libtool、libuuid、libuuid-devel：

```
yum -y install gcc-c++ uuid* libtool libuuid libuuid-devel
```

安装 JDK：

```
chmod +x jdk-6u35-linux-x64-rpm.bin
./jdk-6u35-linux-x64-rpm.bin
```

vi /etc/profile 设置环境变量，在最后面加入：

```
#set java environment
JAVA_HOME=/usr/java/jdk-1_5_0_02
CLASSPATH=.:$JAVA_HOME/lib.tools.jar
PATH=$JAVA_HOME/bin:$PATH
export JAVA_HOME CLASSPATH PATH
```

保存退出，使用命令 java -version 检查是否安装成功。

（3）安装 Zookeeper。

将 Zookeeper 安装包放入系统目录：

```
cp -R zookeeper-3.3.5 /usr/local/
```

为该文件夹添加一个符号链接：

```
ln -s /usr/local/zookeeper3.3.5/ /usr/local/zookeeper
```

vim /etc/profile，设置 ZOOKEEPER_HOME 和 ZOOKEEPER_HOME/bin：

```
export ZOOKEEPER_HOME="/path/to/zookeeper"
export PATH=$PATH:$ZOOKEEPER_HOME/bin
```

使用 zoo_sample.cfg 制作$ZOOKEEPER_HOME/conf/zoo.cfg：

```
cp /usr/local/zookeeper/conf/zoo_sample.cfg /usr/local/zookeeper/conf/zoo.cfg
```

新建两个目录用于 zookeeper 工作时存放日志文件和临时文件：

```
mkdir /tmp/zookeeper
mkdir /var/log/zookeeper
```

至此，Zookeeper 已经安装完成。

（4）安装 ZeroMQ。

进入该软件包的目录：

```
cd zeromq-2.2.0
```

配置安装：

```
./configure
make
make install
```

更新动态链接库：

```
ldconfig
```

ZeroMQ 安装完成。

（5）安装JZMQ。

进入该软件包目录：

```
cd jzmq
```

配置安装：

```
./autogen.sh
./configure
make
make install
```

至此JZMQ安装完成。

（6）安装Storm。

解压Storm软件包：

```
unzip storm-0.8.1.zip
```

如果没有unzip 命令,请安装：

```
yum -y install unzip
```

移动解压后的目录到系统安装目录：

```
mv storm-0.8.1 /usr/local/
```

为该目录添加一个符号链接：

```
ln -s /usr/local/storm-0.8.1 /usr/local/storm
```

vi /etc/profile 为Storm配置环境变量，添加下面三行：

```
#storm
export STORM_HOME=/usr/local/storm-0.8.1
export PATH=$PATH:$STORM_HOME/bin
```

保存退出，到此已完成Storm相关软件在一个节点的安装。

（7）将步骤（1）~（6）在其余3个节点分别执行一遍，但是其中的Zookeeper服务可以选择安装一个或者多个，即步骤（3）可以只在一个节点上操作，也可以在多个节点上操作，本文在192.168.122.101和192.168.122.102两个节点上安装了Zookeeper服务。

8.3.2 Storm 的设置

设置Zookeeper（两个节点均需做如下操作，注：如果是单节点，就不需要以下操作)：

```
vi /usr/local/zookeeper/conf/zoo.cfg
```

文件最后添加一行：

```
server.1=192.168.122.101:2888:3888
server.2=192.168.122.102:2888:3888
```

保存退出，Zookeeper设置完成。

设置Storm（4个节点均需做如下操作）：

```
vi /usr/local/storm/conf/storm.yaml
```

将 storm.yaml 文件中的：

```
# storm.zookeeper.servers:
#     - "server1"
#     - "server2"
```

替换为

```
storm.zookeeper.servers:
    - "192.168.122.101"
    - "192.168.122.102"
```

将

```
# nimbus.host: "nimbus"
```

替换为

```
nimbus.host: "192.168.122.101"
```

添加 Storm 临时文件存放目录：

```
storm.local.dir: "/tmp/storm"
```

另外，可根据节点性能情况适当添加 Supervisor 进程槽端口号，添加几个端口号就表示该节点启动多少个 Supervisor 进程，本文添加了如下 4 个端口号。实际地，可以根据节点性能添加更少或者更多的端口号。

```
supervisor.slot.ports:
 - 6701
 - 6702
 - 6703
 - 6704
```

保存退出，然后新建文件夹/tmp/storm 作为 Storm 运行时存放临时文件的目录。

```
mkdir -p /tmp/storm
```

到此，一个节点上的 Storm 的设置完成，其余节点配置一样，可以将这个文件复制到其他节点的/usr/local/storm/conf/目录中，替换掉以前的 storm.yaml 文件。

8.3.3　Storm 的启动

（1）在节点 192.168.122.101 和 192.168.122.102 上启动 Zookeeper 进程：

```
zkServer.sh start
```

（2）在主节点 192.168.122.101 上启动 Nimbus、Supervisor、UI 进程：

```
storm nimbus &
storm supervisor &
storm ui &
```

UI 进程是一个 Storm 系统的 Web 图形管理进程，UI 进程启动后可以通过浏览器查看 Storm 系统状态。

（3）在子节点 192.168.122.102~192.168.122.104 启动 Supervisor 进程：

```
storm supervisor &
```

现在检测是否安装成功,通过浏览器输入 192.168.122.101:8080 查看，如图 8.5 所示，成功后就可以启动实例进行测试。

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.8.1	2h 14m 57s	4	0	16	16	0	0

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks

Supervisor summary

Host	Uptime	Slots	Used slots
node1	2h 14m 18s	4	0
node2	2h 14m 34s	4	0
node3	2h 14m 9s	4	0
node4	2h 13m 52s	4	0

图 8.5　Storm 的 Web 监控界面

8.4　Storm 使用实例

以上几节介绍了 Storm 的原理和体系架构，本节将通过实例讲解 Storm 的使用方法。

为了让用户尽快掌握 Storm 的使用方法，Storm 的创始人 Nathan Marz 开发了一个让 Storm 用户快速入门的项目——storm-starter，这个项目里有很多适合初学者动手练习的 Topology 示例，如 ExclamationTopology、WordCountTopology、ReachTopology 等，storm-starter 项目详情可登录 GitHub 官网进行查看。

使用 storm-starter 中的 Topology 之前，首先需要安装编译该项目的软件。编译 storm-starter 项目有两种方法，一种是使用 Leiningen，另一种是使用 Maven。Leiningen 是一个用于自动化构建 clojure 项目的工具，而 Maven 是一个基于项目对象模型（POM）的项目管理工具，这两种工具都可以用于项目管理。

8.4.1　使用 Maven 管理 storm-starter

在 Storm 中，提交 Topology 只需在主节点上进行，因此我们只需在主节点上安装 Maven。

1. Maven 的安装

回到主节点 192.168.122.101，使用用户 storm，并切换到 storm 目录：

```
su - storm
cd storm
```

解压 maven 包：

```
unzip apache-maven-3.0.5-bin.zip
```

将解压后的目录存放到系统目录：

```
cp apache-maven-3.0.5-bin /usr/local/
```

重命名该目录：

```
mv /usr/local/apache-maven-3.0.5-bin /usr/local/maven
```

为 Maven 使用添加环境变量：

```
vi /etc/profile
```

添加如下内容：

```
#maven
M2_HOME=/usr/local/maven
PATH=$PATH:$M2_HOME/bin
export M2_HOME PATH
```

保存退出。

使新加环境变量生效：

```
source /etc/profile
```

到此，关于 Maven 的安装完成，接下来就是如何使用。

2．使用 Maven 管理 storm-starter

进入 storm-starter 目录：

```
cd storm-starter
```

执行编译命令将该项目打包为 jar 文件：

```
mvn -f m2-pom.xml package
```

执行完后，可以发现 storm-starter 目录下的/arget 目录中多了两个 jar 文件，一个是带有依赖包的 jar 文件，一个是不带有依赖包的 jar 文件，我们需要的是带有依赖包的这个。

3．提交 storm-starter 中的 Topology

进入/target 目录：

```
cd target
```

提交 WordCount 的 Topology：

```
storm jar storm-starter-0.0.1-SNAPSHOT-jar-with-dependencies.jar
storm.starter.WordCountTopology wordcountTop
```

提交后，可以在浏览器页面看到提交的 wordcountTop 如图 8.6 所示。

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.8.1	2h 32m 34s	4	3	13	16	26	26

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
wordcountTop	wordcountTop-1-1389275434	ACTIVE	6s	3	26	26

Supervisor summary

Host	Uptime	Slots	Used slots
node1	2h 31m 59s	4	1
node2	2h 32m 11s	4	0
node3	2h 31m 44s	4	1
node4	2h 31m 33s	4	1

图 8.6　启动 wordcountTop

storm 提交 Topology 命令格式如下：

```
storm jar all-my-code.jar backtype.storm.MyTopology arg1
```

all-my-code.jar 为要提交的 jar 包名，backtype.storm.MyTopology 为要执行的该 jar 包中的 Topology 名，arg1 表示要为提交的 Topology 取的运行后的名字，如果为空它会使用该 Topology 的默认名。

以上提交的 Topology 表示提交的是 storm-starter 中的 WordCountTopology，它运行后的名字为 wordcountTop。

8.4.2　WordCountTopology 实例分析

WordCountTopology 是 Storm 的典型使用案例，体现出 Storm 对数据流进行实时处理的特性，其设计模型和工作过程在本章 8.2.1 节进行了讲解，本节将详细分析 WordCountTopology 的编码，使大家对 Storm 的运行过程有深入的了解，进而可以开始编写自己的 Topology。

WordCountTopology 的主要代码段如下：

```
TopologyBuilder builder = new TopologyBuilder();//创建一个 TopologyBuilder 对象 builder 用于建造 wordcount 拓扑
/* 设置一个 Spout 组件，用于随机读取流式数据，组件的名称为 spout，组件完成任务的功能类为 RandomSentenceSpout()，组件的并行数为 5*/
builder.setSpout("spout", new RandomSentenceSpout(), 5);
/*设置 Bolt 组件，用于对读取的句子进行切分，组件的并行数为 8*/
builder.setBolt("split", new SplitSentence(), 8)
        .shuffleGrouping("spout");//设置接收消息的分组方式为随机分组，接收消息来源于 Spout 组件
/*设置 Bolt 组件，用于对切分后的单词进行统计，组件的并行数为 12*/
builder.setBolt("count", new WordCount(), 12)
        .fieldsGrouping("split", new Fields("word"));//设置接收消息的分组方式为字段分组
```

首先，构造一个 Topology 需要一个 TopologyBuilder 类，设置好一个 TopologyBuilder 对象

后就可以开始构建该 Topology。Topology 中包含两类组件：Spout 和 Bolt，分别使用 setSpout() 和 setBolt()方法设置，这两个函数的参数相同，均包含组件名、组件完成任务的类和该组件的并行任务数目，其中组件完成任务的类是编写 Topology 需要主要编写类，该类主要包括 Spout 类和 Bolt 类。

这里，编写自己的 Spout 类和 Bolt 类主要继承 Storm 中的 Spout 基类和 Bolt 基类，并重写基类中的方法来实现。

在 WordCountTopology 中，Spout 组件完成的功能类为 RandomSentenceSpout()，该类继承于 Spout 基类 BaseRichSpout 类，RandomSentenceSpout 类实现代码如下：

```
    public class RandomSentenceSpout extends BaseRichSpout {
        SpoutOutputCollector _collector;//成员变量 1，SpoutOutputCollector 类型，用于发送消息
        Random _rand;    //成员变量 2，Random 类型，用于生成随机数
        @Override
        //spout 组件初始化
        public void open(Map conf, TopologyContext context, SpoutOutputCollector
collector) {
            _collector = collector;
            _rand = new Random();
        }
        @Override
        //读入数据流
        public void nextTuple() {
            Utils.sleep(100);//睡眠 100 毫秒
            String[] sentences = new String[] {
                "the cow jumped over the moon",
                "an apple a day keeps the doctor away",
                "four score and seven years ago",
                "snow white and the seven dwarfs",
                "i am at two with nature"};//定义一个存放了 5 个字符串的 String 数组
            String sentence = sentences[_rand.nextInt(sentences.length)];//随机从数组
中读取一个字符串
            _collector.emit(new Values(sentence));//发送该字符串
        }
        @Override
            //用于保证发送的消息被完整的处理
```

```
    public void ack(Object id) {
    }
    @Override
        //当消息没有被完整处理时，决定如何处理这条消息
    public void fail(Object id) {
    }
    @Override
        //定义该组件发送的消息域名
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

以上代码包含 5 个方法，其中最重要方法是 netTuple()，该函数用来决定如何读入数据流。该方法中实现的主要功能是每隔 100 毫秒随机地从存放 5 条句子的 String 数组中取一条数据，并发送出去。

WordCountTopology 的 Bolt 组件包含 split 和 count。实现 splilt 组件功能的类为 SplitSentence()，实现代码如下：

```
public static class SplitSentence extends ShellBolt implements IRichBolt {
        public SplitSentence() {
            super("python", "splitsentence.py");//使用父类构造函数，表明该方法使用 Python
语言实现，实现文件名为 splitsentence.py
        }
        @Override
            //定义消息输出域名
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
        @Override
            //获取组件配置信息
        public Map<String, Object> getComponentConfiguration() {
            return null;
        }
    }
```

该类演示了使用第三方语言实现组件的方法，当前 Storm 已经支持的第三方语言适配器为 Python 和 Ruby。我们可以通过查看 Javadoc 中的 ShellBolt 说明来了解如何通过 Java API 来使用第三方语言实现组件功能。splitsentence.py 实现代码如下：

```
import storm
class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup)://定义一个将字符串切分为单词的函数
        words = tup.values[0].split(" ")//以空格为划分切分字符串
        for word in words://遍历存放单词的元组
          storm.emit([word])//将单词发送出去

SplitSentenceBolt().run()
```

下面讲解如何实现 count 组件。count 组件功能类为 WordCount，它继承 Storm 实现 Bolt 基类 BaseBasicBolt，其实现代码如下：

```
public static class WordCount extends BaseBasicBolt {
        Map<String, Integer> counts = new HashMap<String, Integer>();//定义一个
Map<String,Integer>类型变量用来保存中间结果
        @Override
    //Bolt 组件中的主要执行方法，它的原型来自于父类 BaseBasicBolt，实现内容由我们自己完成
        public void execute(Tuple tuple, BasicOutputCollector collector) {
            String word = tuple.getString(0);//获取接收到的数据队列中的第一个单词
            Integer count = counts.get(word);//从 counts 中获取该单词的 value 值
            if(count==null) count = 0;//如果为空，则表示这个单词第一次到来，设置 value 值为 0
            count++;//在原基础上加 1
            counts.put(word, count);//更新 counts 变量
            collector.emit(new Values(word, count));//将当前单词统计数据发送出去
        }
        @Override
            //定义该组件域
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word", "count"));
        }
    }
```

在 count 组件实现中，通过维持一个中间变量 count 来保存中间数据，每次更新完<word, count>键值对都会实时发送数据，我们可以非常及时地了解当前单词出现的次数。

组件与组件之间通过消息传递的方式交互数据，数据传递的方式在 setSpout()和 setBot()方法后面进行设置。setSpout()和 setBolt()返回值类型为 BoltDeclarer 类，该类定义了设置该组件接收消息的如下 6 种方法。

（1）随机分组（shuffleGrouping）：随机分发元组到 bolt 的任务，保证每个任务获得相等数量的元组。

注释

元组：数据项目组成的列表。

（2）字段分组（fieldsGrouping）：根据指定字段分割数据流，并分组。例如：根据“word”分组，相同“word”的元组总是分发到同一个任务，不同“word”元组可能分发到不同任务。

（3）全部分组（allGrouping）：元组被复制到 bolt 的所有任务。

（4）全局分组（globalGrouping）：全部流都被分到 bolt 的同一个任务，实际中往往是被分配到 ID 最小的任务。

（5）无分组（noneGrouping）：不需要关心流如何分组。现在的无分组方式等同于随机分组方式。

（6）直接分组（directGrouping）：元组生产者决定元组由哪个元组消费者接收。

本节的 WordCountTopology 中 split 组件接收消息的方式设置为随机分组，count 组件接收消息的方式设置为字段分组，字段域名即 split 组件中声明的“word”，即表示接收该组件中的数据。

另外，该 Topology 中所有的数据都在内存中，我们如果想看到数据，可以通过把 count 组件最后发送的数据写入文件，这样我们就可以在文件中查看这些实时结果。

练习题

1. Storm 的三进程架构包括________、________、________。
2. 在 Storm 中每实现一个任务，用户需要构造包含________、________组件的拓扑。
3. 动手搭建 Storm 开发环境。

PART 9

第 9 章 服务器与数据中心

9.1 数据中心的发展历史

1. 数据中心的定义

数据中心是用于存放计算机系统和与之配套的网络、存储等设备的综合系统，数据中心需要具备冗余的数据通信连接、环境控制设备、监控设备以及各种安全装置。谷歌在其发布的《The Datacenter as a Computer》一书中，将数据中心定义为：多功能的建筑物，能容纳多个服务器以及通信设备，这些设备被放置在一起是因为它们具有相同的对环境的要求以及物理安全上的需求，并且这样放置便于维护，而并不仅仅是一些服务器的集合。

2. 数据中心的发展历程

第一阶段：巨型机时代。

20 世纪 60 年代以前，计算机使用大量的真空管作为计算部件，部件之间需要大量的线缆连接，计算机的操作和维护很复杂，其总功率很大，往往超过 100 万千瓦/时，需要专门的供电和制冷系统，计算机的体积庞大，需要一两百平方米的房间来存放，而且当时的计算机主要用于军事用途，单独存放巨型计算机的房间就是今天“数据中心”的雏形。

- 数据中心的鼻祖——大型机 ENIAC

在第二次世界大战期间，美军为了研制新型武器，在马里兰州的阿伯丁设立了“弹道研究实验室”。但是研制新型机所需的计算量让研究人员大为头疼，200 名计算快手不停地计算，但是效率还是很低，他们迫切需要一种新型的计算器来完成这些繁重的计算。当他们正在为这一问题头疼的时候，宾夕法尼亚大学莫尔电机学院的莫克利博士提出了试制第一台电子计算机的设想。

1946 年 2 月 14 日，人类收到了一个庞大的情人节礼物，ENIAC（Electronic Numerical Integrator And Computer，电子数字积分计算机）正式诞生，这是一个长 30.48 米，宽 1 米，重达 30 吨，功率超过 174 千瓦/时，体内安装了 18000 个电子管，花费 48 万美元。它能在 1 秒内进行 500 次加法运算或 400 次乘法运算，运算速度是当时最快的继电器计算机的 1000 倍，是手工计算的 20 万倍，ENIAC 是世界上第一台真正意义的数字电子计算机，如图 9.1 所示。

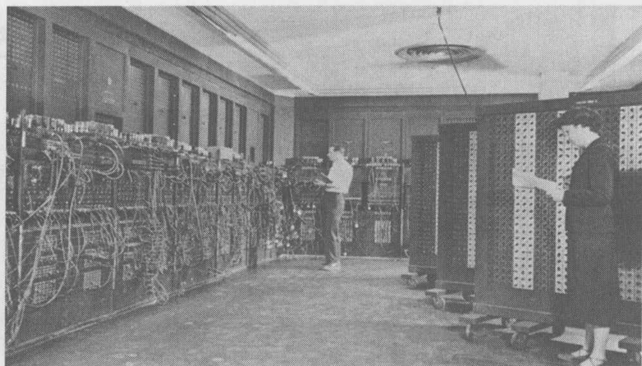


图 9.1 世界上第一台真正意义的数字电子计算机——ENIAC

这台计算机的总占地面积约 170 平方米，采用电子真空管作为运算部件，电子射线管作为存储部件，没有内存储器。其运行和维护很复杂，每执行一个新的运算任务需要像搭积木一样把运算部件进行重新搭配一遍，其电子管平均 15 分钟就要烧坏一只，因此需要有专人 24 小时进行值班维护。ENIAC 被公认为是"数据中心"的鼻祖。

- 第一台晶体管计算机——IBM 7070

到了 20 世纪 60 年代，计算机的真空管部件逐渐被晶体管所取代，计算机的体积大幅减小，运算速度快速提升、可靠性大幅增强，同等性能的计算机造价比晶体管时代大幅下降。IBM 的 7070 型大型机（如图 9.2 所示）是世界上第一台晶体管计算机，这台计算机使用穿孔卡片，有 32KB 的内存，用户数据在内存和一台磁鼓之间切换。美国航空公司和 IBM 联合开发了 Sabre 航空订票系统，该系统使用了两台 7070 型大型机，放在一个单独设计的数据中心，每天处理 84000 个航空电话业务，自此计算机开始用于商业用途，企业级数据中心开始出现。



图 9.2 IBM 7070 型大型机

第二阶段：微型计算机/PC 时代。

到了 20 世纪 70~80 年代，小型计算机产业发展迅速，计算机朝着体积更小、性能更强的方向发展。1973 年世界上第一台具有桌面图形界面和鼠标键盘的计算机 Xerox Alto（如图 9.3 所示）问世，Xerox Alto 成为此后苹果公司 Macintosh 和 SUN 公司 Workstation 的灵感来源，也是第一个所见即所得应用程序和第一个真正的 Smalltalk 集成开发环境的硬件平台。1988 年 CRAY Y-MP 巨型计算机（如图 9.4 所示）的面市推动了高性能数据中心的发展，当时美国国家大气研究中心的计算中心就采用了这种机型。



图 9.3 Xerox Alto

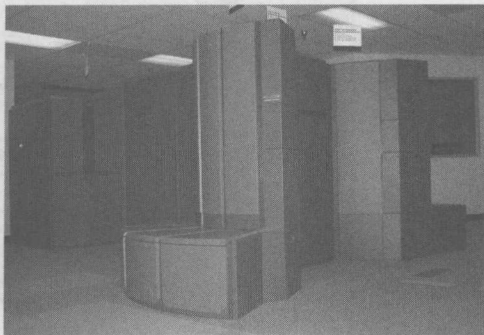


图 9.4 CRAY Y-MP

这段时间，很多企业开始使用小型化的计算机来进行公司业务操作和数据处理，业务流程和业务数据与计算机的融合加深，许多公司开始将多台计算机放置在一个房间中来方便维护、管理。

到了 20 世纪 90 年代，PC（Personal Computer）时代来临，随着 Linux 和 Windows 操作系统的出现，PC 的使用快速普及，PC 开始出现在计算机房中，并用价格昂贵的网络设备通过 C/S 模式的分时操作系统供多用户共享计算机资源。性能快速提升的数据中心在不断小型化的同时开始通过网络为多用户提供共享资源和计算服务，现代数据中心的雏形开始显现。

第三阶段：互联网时代。

20 世纪 90 年代中期，随着互联网浪潮的到来，数据中心出现了真正的大发展，很多公司都需要高速、稳定的互联网连接以保障企业的网络业务，这个时期很多公司修建了大规模的互联网数据中心（IDC，Internet Data Center）。

第四阶段：云计算、大数据时代。

巨型机、微型机和互联网是数据中心发展历程中的关键性的节点。在巨型机时代，计算是集中式进行的，所有计算均在巨型机上进行，科学家们根据具体的计算任务进行操作，巨型机的主要功能是科学计算。到了微型计算机/PC 时代，企业的业务系统等应用部署在了计算机上，应用系统在数据中心处于核心地位；在互联网时代，应用系统是数据中心的服务器核心，这个阶段应用系统在设计的时候一般固定运行在若干台托管的服务器上，当业务系统的压力变化时，无法动态、实时地对服务器集群规模进行调整。

云计算、大数据时代对于数据中心的安全性、稳定性、集群管理、能耗问题、环境影响等方面提出提出了更高的要求。Google 的数据中心数云计算大数据时代典型的数据中心，如图 9.5 所示。

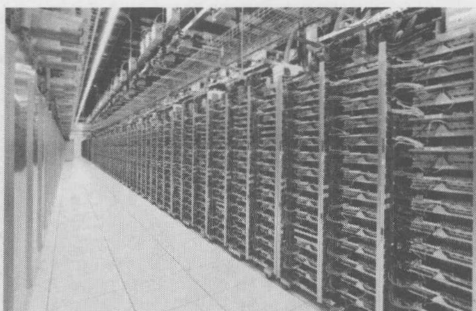


图 9.5　Google 数据中心

目前，谷歌、微软公司在全球各地的数据中心服务器总量均超过 100 万台，亚马逊的联网计算机数量为 15.8 万台。自 2006 年谷歌建立自己定义的数据中心以来，其在互联网基础设施方面的投资已经超过 210 亿美元，2012 年以来，谷歌每个季度的投资均超过 10 亿美元。

谷歌一般会选择在电力成本低廉、绿色能源丰富、水资源丰富、地域开阔、与其他数据中心距离合理的地方来新建数据中心。其服务器的规模占全球服务器总量的 3%，但只消耗了全球数据中心 1%的电力，可再生能源的使用量展期总电力消耗的近 30%，这得益于谷歌的数据中心节能环保技术。谷歌将数据中心的冷通道温度保持在 27℃，并使用外部空气冷却其数据中心，而不是使用耗能的冷却系统。

谷歌的服务器都是由其自行设计，减少不需要的零组件，减少不必要的部件能耗，减少风扇数据，提高能源使用率效率。

3．数据中心的组成

数据中心主要由基础设施、硬件设施、基础软件、管理支撑软件构成，各部分的主要组成如此下：

（1）基础设施：机房、装修、供电(强电和 UPS)、散热、布线、安防等部分；

（2）硬件设施：机柜、服务器、网络设备、网络安全设备、存储设备、灾备设备等；

（3）基础软件：操作系统、数据库软件、防病毒软件等；

（4）管理支撑软件：机房管理软件、集群管理软件、云平台软件、虚拟化软件等。

4．云计算大数据时代的数据中心发展趋势

（1）设备小型化。

（2）管理智能化。

（3）集中化建设，可弹性扩展。

（4）数据的价值凸显。

（5）绿色节能化。

9.2 数据中心的基本单元——服务器

服务器（Server）是指运行操作系统、数据库系统、Web 系统等软件系统为网络上其他终端提供服务的硬件设备。服务器通常都会采购专用的 CPU，与 PC 相比，服务器有更高标准的主板和电源、以及专用的带纠错功能的高速内存（ECC 内存）和专用的硬盘（SAS 硬盘、FC 硬盘、SSD 硬盘）。

目前常见的服务器从基础构架上分为使用 RISC（精简指令集）CPU 的专用服务器（中型机、小型机），这类服务器主要用于对浮点运算性能较高的应用场景，以及使用 CISC（复杂指令集）CPU 的通用服务器（X86 服务器）。Intel 公司的 Xeon 系列和 AMD 公司的 Opteron 系列都使用了 CISC 指令集，因为 Xeon 和 Opteron 与 PC 机的 CPU 都使用了 X86 架构。所以这类服务器通常被称为 X86 服务器。

X86 服务器可从机箱结构和外形、节点密度、支持的 CPU 数量分类。

（1）按机箱结构和外形分类：塔式服务器、机架式服务器；

（2）按节点密度分类：单节点服务器、多节点（高密度）服务器；

（3）按支持的 CPU 数量分类：单路服务器、双路服务器、4 路服务器、8 路服务器、16 路服务器、32 路服务器。

下面对这 3 种分类方式进行详细讲解。

1．按机箱结构和外形分类

（1）塔式服务器。

塔式服务器的外形如图 9.6 所示，塔式服务器机箱从体积上可以分为全塔式、中塔式、mini 塔式，塔式服务器占用的体积比较大，通常数据中心或云计算中心不采用塔式服务器。

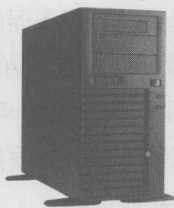


图 9.6　标准的塔式服务器

（2）机架式服务器。

对于建设费用高昂的数据中心来说，数据中心空间利用率非常重要，塔式服务器的空间利用率较低，不适合用于数据中心，机架式服务器应运而生，如图 9.7 和图 9.8 所示。

机架式服务器采用了与交换机一样的长方体结构，美国电子工业协会（EIA）制定了统一的标准尺寸，标准宽度（两端上架孔距）固定为 470mm，标准深度为 650mm，在高度方面，EIA 推出了一个专用计量单位“U”（Unit 的缩略语）。1U=1.75 英寸=44.5mm。机架式服务

器在高度上分为 1U/2U/3U 等。后面要讲到的刀片式服务器（刀片机）也属于机架式服务器。



图 9.7 1U 机架式服务器

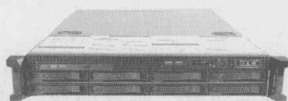


图 9.8 2U 机架式服务器

2．按节点密度分类

节点密度是单台服务器内置的独立系统单元数目，即单台服务器内置的主板数目。内置一个系统单元的服务器称为单节点服务器，单节点服务器是我们平时最常见的服务器类型；内置两个系统单元的服务器称为双节点服务器，通常被称为“双子星”服务器。以此类推还有“四子星”、“八子星”等，节点密度一般为偶数。

单台服务器的密度在四节点及以上的，又被称作高密度服务器。刀片机也是高密度服务器的一种，相比“四子星”、“八子星”等高密度服务器，刀片机在电源模块和网络模块等方面拥有更高的集成密度，目前主流的刀片机可达 7U/16 刀及以上的密度。

图 9.9~图 9.12 分别是双子星、四子星、18 子星、刀片机（7U/14 刀）。

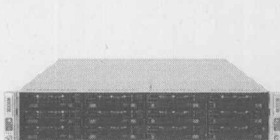


图 9.9 双子星服务器

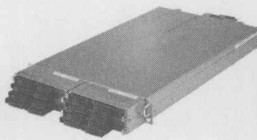


图 9.10 四子星服务器

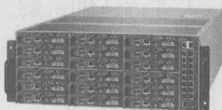


图 9.11 18 子星服务器

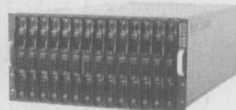


图 9.12 刀片机

3．按支持 CPU 数量进行分类

服务器所指的“单路”、“双路”是指单台服务器的主板上所能支持安装 CPU 的数量，一般为偶数个 CPU 协同工作，用以得到更高的单机（单元）计算性能。

只支持一颗 CPU 的服务器称为单路服务器；支持两颗 CPU 的服务器称为双路服务器；支持四颗 CPU 的服务器称为四路服务器；双路及以上的服务器统称为多路服务器。图 9.13~图 9.15 所示分别为单路服务器、双路服务器、四路服务器。

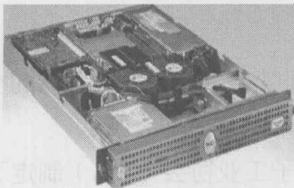
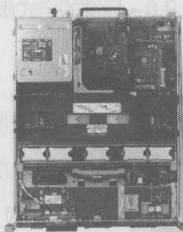
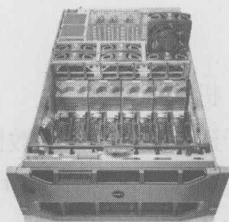


图 9.13 单路服务器



9.14 双路服务器



9. 15 四路服务器

9.3 数据中心选址

数据中心的选址是数据中心建设的早期重要工作，数据中心的使用年限往往会超过 20 年，数据中心的建设、运行、维护涉及对于地质条件、气候环境、电力供给、网络带宽、人力资源等条件，需要综合考虑诸多因素。

地质环境：大型数据数据中心在选址的时候一般倾向选择建设在地质条件比较稳定，地震、沉降等自然灾害较少的地区，减少自然灾害等不可抗力对数据中心运行的影响概率。

气候条件：气候条件对于数据中心的建设、运行成本有直接影响，建设在寒冷地区的数据中心的数据中心与建设在炎热地区的数据中心相比，用于制冷的电力成本大幅降低，同时其制冷系统的建设级别和造价相对较低。Google 在比利时、芬兰等寒冷地区建设了自己的数据中心，尤其是建设在比利时的数据中心基本全年性地采用无需制冷剂的自由冷却方式对数据中心进行降温，制冷系统造价和电力成本非常低。

电力供给：数据中心是电力消耗的大户，在美国，数据中心的能耗已经超过美国全国用电量的 1.5%；2012 年全球数据中心的总能耗已超过 300 亿瓦，相当于 30 座核电站的发电量；单个数据中心的能耗已经上升到千万瓦的级别，数据中心在选址时必须要考虑当地的电力供应能力和电力成本。

网络带宽：网络带宽是数据中心为用户提供服务的核心资源，网络带宽直接影响用户的请求响应及时性，是数据中心选址考虑的重要因素，需要选择网络带宽条件较好的骨干网节点城市。

水源条件：目前先进的数据中心的冷却系统经常采用水冷系统进行蒸发冷却，用水量巨大，例如，微软公司的圣安东尼奥数据中心每年需要消耗 38 万吨水用于制冷，数据中心选址时需要考虑当地的水源供给情况。

人力资源：数据中心在选址时需要选择在能够提供必要的数据中心的建设、维护、运营等人力的地区。

9.4 数据中心的能耗

本章前面部分提到，数据中心的硬件部分由机柜、服务器、网络设备、网络安全设备、存储设备、灾备设备等组成，数据中心的能耗控制可以分为数据中心级、节点级、器件级，如图 9.16 所示。

数据中心级：从较为宏观的数据中心级别来看，制冷系统所消耗的能源占数据中心总体能耗的比重较大，制冷系统的规划和性能对于数据中心整体能耗的影响很大。从集群软件角度来看，数据中心的任务调度和负载均衡系统是影响集群性能发挥的重要因素。

节点级：节点级的能耗控制主要在于根据节点的负载状况动态调整处于工作状态的节点数量。

器件级：CPU、内存、硬盘等器件的能耗主要通过调整工作电压和频率的方式来控制。



图 9.16 数据中心的能耗结构

数据中心的能耗通常是通过 PUE、DCIE、IT 设备能效比等参数来进行评估。

1．PUE

PUE(Power Usage Effectiveness)由美国绿色网格联盟(The Green Grid)于 2007 年提出，是业界公认的测量数据中心能耗的主要指标之一，其定义如下：

$$\text{PUE} = \frac{\text{数据中心整体能耗}}{\text{IT设备能耗}}$$

IT 设备的能耗为数据中心计算、存储、网络等核心设备的总能耗，包含服务器、网络设备、存储设备等；数据中心整体能耗为 IT 设备能能耗、制冷设备能耗、电源能耗、控制仪表等设备的能耗的综合。

PUE 值表示数据中心的总能耗为 IT 设备能耗的倍数，其值越小表示用于数据中心计算、存储等核心设备的运行的能耗比例越大，数据中心的能源效率越高。例如，PUE=3 时，数据中心总能耗为 IT 设备能耗的 3 倍，服务器等 IT 设备每消耗 1 度电，空调等其他设备就要消耗 2 度电，数据中心总体能效较低；当 PUE= 1 时，数据中心的所有电能都用于 IT 设备的运行，没有其他的能量损耗，是 PUE 的理论最小值。

Google 公司的拥有数以百万计的庞大服务器集群，其数据中心建设采用一系列先进的建设技术，其全年平均 PUE 值为 1.12，最优值为 1.06，远优于全球 PUE 值的平均值 1.8~1.89。我国的数据中心 PUE 值相对较高，全国数据中心 PUE 平均值为 2.5，百度 M1 云计算中心的 PUE 值最低，全年平均值为 1.35，最优值为 1.18，是国内能效最高的数据中心。

例题：

数据中心 A 有 100 个机柜，每个机柜有 5 台 2U/18 刀的刀片式服务器，运行时总功率为 500kW，IT 设备功率 250 kW，该数据中心的 PUE=500/250=2。

数据中心 B 由 200 个机柜，每个机柜有 6 台机架式 2U/4 刀的机架式服务器，运行时功率为 700 kW，IT 设备功率为 300 kW，该数据中心的 PUE=700/300=2.33。

以上表明数据中心 A 的 PUE 值低于数据中心 B，数据中心 A 的能效更高。

2．DCIE

DCIE(Data Center Infrastructure Efficiency)是数据中心能耗评估的另一公认指标，是由美国绿色网格联盟于 2007 年提出，用于表示数据中心 IT 设备用电占总用电量的比例，其定义如下：

$$\text{DCIE} = \frac{\text{IT设备能耗}}{\text{数据中心整体能耗}}$$

DCIE 是 PUE 的倒数，其数值小于 1，越接近于 1 表明数据中心能源用于 IT 设备的比例越高，数据中心的能源效率越高。

3．IT 设备的能效比

IT 设备自身的能效也是数据中心能效的一个重要指标，其定义如下：

$$\text{IT设备的能效比} = \frac{\text{IT设备每秒的数据处理流量}}{\text{IT设备的能耗}}$$

这里的 IT 设备指的是服务器、存储等设备，IT 设备的能效比越高 IT 设备每消耗单位电能所能处理、存储和交换的数据量越大；执行相同的计算、存储、通信任务 IT 设备的能效比越高消耗的能量就越低，设备与周围环境的热交换就越少，这样可以降低数据中心 UPS 和空调系统的设计容量，进一步降低数据中心的能耗，提高数据中心的能效。

练习题

1. 数据中心的发展经历了________、________、________、________四个阶段。
2. 数据中心的选址主要考虑________、________、________、________等因素。
3. 数据中心的主要组成部分有________、________、________、________。
4. PUE 的定义为________________________。
5. DCIE 的定义为________________________。

PART 10 第 10 章 云计算大数据仿真技术

计算机仿真是将现实生活中的系统，用计算机软件的方法在计算机中建立虚拟的替代物，以方便人们研究系统各个方面的性质。比如，我们可以使用计算机软件对不同气候条件下的导弹的发射进行建模，模拟导弹的运动轨迹，我们可以对不同型号的导弹在不同的动力系统条件下的运动轨迹进行反复实验。使用计算机软件的仿真技术具有经济、安全、可重复和不受气候、场地、时间限制的优势，是理论推导和科学实验之外人类认识自然、改造自然的重要手段。

本书前面几章讲解了基于虚拟化的云计算技术、基于集群的云计算技术和云计算数据中心的相关知识，基于这些技术目前已经有很多的系统级、算法级和应用级的研究展开，这些开发和研究大多需要仿真平台。比如，技术研发人员对大规模集群的资源调度、负载均衡、集群拓扑等展开研究，如果在物理机上进行实验，必然需要消耗大量的服务器、网络设备资源，实验环境的准备、实验数据的采集、实验方案的调试很不方便，同时成本很高，使用仿真系统是一个很好的解决方案；对于数据中心的建设和运营人员来说，数据中心的能耗测算和经济测算非常重要，需要在项目建设之前进行预估，无法在实际的平台上进行测算，展开研究需要先在仿真实验平台上进行实验。

本章的主要内容包括云计算仿真软件 CloudSim 和云计算系统相空间模型，通过仿真软件和仿真模型使读者快速掌握云计算仿真的基础知识。

10.1 用参数定义物理设备进行仿真

在仿真系统中，我们一般将实体的参数提取出来，用变量、对象、数组来定义现实中的事物在计算机系统中构建被仿真对象。

服务器是数据中心的主要组成部分，我们可以将服务器的计算性能、CPU 核数、硬盘大小、内存大小、网络带宽等主要参数提取出来，构建服务器对象。

下面以一个由 4 个虚拟机节点组成的集群为例，说明如何使用参数来定义物理节点，集群的参数信息如表 10.1 所示。

表 10.1 虚拟机性能参数

	计算能力 (MIPS)	CPU 核数	硬盘大小 (MB)	内存大小 (MB)	网络带宽 (MB)
节点 1	300	2	10000	512	1000
节点 2	300	2	10000	512	500
节点 3	150	1	5000	256	500
节点 4	150	1	5000	256	1000

我们可以用数组来定义：

```
double node[4][5]={300,2,10000,512,1000;
                   300,2,10000,512,500;
                   150,1,5000,256,500;
                   150,1,5000,256,1000};
```

对硬盘、内存等其他设备也可以如此进行定义，比如，硬盘可以用品牌、型号、尺寸、容量、转速、传输速度等参数来定义，内存可以用品牌、型号、容量、速度、电压等参数来定义。在对数据中心的经济模型进行仿真时，我们需要对 CPU 核单价、硬盘空间单价、内存空间单价、网络带宽单价等进行定义，确定销售经济模型。同时，需要定义数据中心能耗费用、维护费用、人员成本等，从而综合确定数据中心的经济模型，对数据中心建成后的营收情况进行预测。

10.2 云计算仿真系统——CloudSim

10.2.1 CloudSim 基础

1. CloudSim 简介

CloudSim 是澳大利亚墨尔本大学云计算与分布式系统实验室开发的一种通用、可扩展的云计算仿真框架，也是一个云计算仿真工具集，提供了用于描述数据中心、虚拟机、应用、用户、计算资源和管理策略等核心类。

对海量集群资源的模拟仿真一直是计算机领域的研究课题。在网格计算时代出现了很多仿真平台，如 GridSim、SimGrid、OptorSim、GangSim 等，其中 GridSim 的开发团队也是澳大利亚墨尔本大学云计算与分布式系统实验室。GridSim 等网格计算仿真软件没有将云计算体系中的 SaaS、PaaS、IaaS 层抽象出来，也没有虚拟化模型和资源管理模型，CloudSim 继承了 GridSim 的编程模型，弥补了网格计算模拟软件的不足。

基于 CloudSim 云计算仿真器，我们不仅能够很方便地搭建可控的云环境进而对系统的资源调度和负载均衡策略进行建模和测试，还可以对云应用进行建模和测试。研发人员根据测评

结果未有针对性地调整性能瓶颈。与此同时，CloudSim 对云系统建立了价格模型和能耗模型，帮助服务提供商制订出更加合理的价格策略和节能机制。

用户可以使用 CloudSim 提供的组件进行编程，构造自己的应用场景，也可以扩展或者自己编写类来进行仿真，使用起来非常灵活。这一点与针对特定使用场景的仿真系统不同，针对特定使用场景的仿真系统在使用的时候只需填写参数即可使用，无需编程，但无法灵活地构建使用场景。

CloudSim 是使用 Java 语言开发的，用户只需掌握 Java 语言的用法和云计算的相关知识，即可建立云计算模型进行仿真。仿真平台是个模拟器，并不能运行真实的云计算平台上的应用程序。

CloudSim 在物理主机和虚拟机两个层面进行资源分配。物理主机中构建的所有虚拟机共享物理资源，由 CloudSim 中的 VmScheduler 负责资源的分配；CloudSim 中仿真的任务称为 Cloudlet，集群中的虚拟机有大量的 Cloudlet 需要资源，由 CloudSim 中的虚拟机资源调度器 CloudletScheduler 负责资源的分配。

2．为什么要使用 CloudSim

对于技术研发人员来说，大规模集群的资源调度、负载均衡、集群平台，集群拓扑等研究如果在物理机上进行，需要大量的服务器、网络设备资源，实验环境的准备、实验数据的采集、实验方案的调试很不方便、成本很高，需要先在仿真实验平台上进行实验。

对云应用服务的测试也会比较麻烦，主要表现在：

（1）应用服务商直接将应用部署到云平台上之后再进行测试，无疑会带来额外的成本开销。一旦应用程序接入云平台就必须要缴纳相应的费用，这样在应用没有任何经济效益的情况下就产生了额外的费用，对于 SaaS 提供商来说是不经济的；

（2）实际运行的云平台环境（IaaS、PaaS）是不可控的，整个互联网环境时而拥塞，时而清闲，从而导致了云平台资源使用的无规律性和不可再现性，不利于应用的重复测试。

3．CLoudSim 的特点

（1）能够在一台 PC 上建模和仿真大规模云计算基础设施，如数据中心、物理主机等；

（2）支持用户任务以及服务代理的建模和仿真；

（3）支持对云计算环境中的网络环境进行建模；

（4）有效地利用虚拟化引擎，帮助在数据中心节点上创建、管理和销毁多个虚拟节点；

（5）可以灵活地在基于时间共享和基于空间共享的虚拟化策略之间进行切换；

（6）支持对云数据中心的能耗行为进行建模和仿真；

（7）可以方便地建立云平台资源的价格策略，包括存储价格、带宽价格等；

（8）能够模仿多个云厂家之间进行透明交易，包括任务迁移、存储迁移、价格协商等。

10.2.2 CloudSim 体系结构

CloudSim 的多层体系架构如图 10.1 所示。

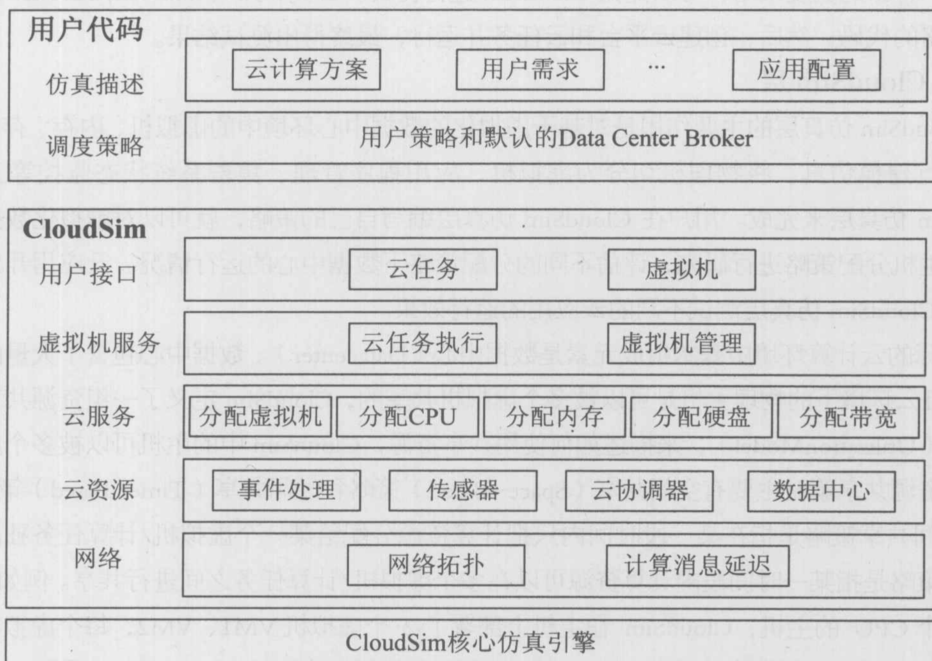


图 10.1 CloudSim 分层体系架构

1．用户代码层

用户代码层处于系统的上层，包含仿真描述和调度策略，用户在这一层定义云计算方案、用户需求，进行应用配置，同时云应用开发人员可以生成工作流请求，根据用户的配置进行云计算场景的强力测试。

（1）仿真描述。

对于云服务使用者来说，他们需要测试应用程序在特定云平台上的服务性能，或者测试应用程序需要占用多少云资源，只需创建与特定云平台类似的虚拟云平台，并按应用程序的需求（如带宽、内存等）创建对应的云任务（在 CloudSim 中云任务被定义为 Cloudlet）。之后，就可以让云任务运行在虚拟的云平台上最终得到测试结果。

比如，一个亚马逊云平台的使用者，想在其上部署一个网络硬盘的应用程序，希望估算需要租用多少服务，可以使用 CloudSim 进行仿真。首先，使用 CloudSim 建立一个虚拟的亚马逊云平台；然后，在其上建立一定数量的虚拟机资源对应某一云服务性能；最后，按照自己的预期生成云服务（比如，需要多大的硬盘、带宽、内存等），使其运行在之前建立的虚拟的云服务上得出测试结果。

（2）调度策略。

从云服务提供者的角度，服务提供者想测试云平台任务调度策略是否合理，或者服务商提

出一种新的任务调度策略，在使用之前需要对其进行测试。测试的重心相较于 CloudSim 就不一样了，测试的步骤需要先实现自定义的任务调度策略（主要是更改数据中心代理 DatacenterBroker）。比如，亚马逊的用户发现当前的任务调度策略没有发挥最好的作用，设计实现了一种新的调度策略，可以先在 CloudSim 进行仿真。首先，改写 DatacenterBroker 的任务调度策略的代码；然后，创建云平台和云任务并运行，最终得出测试结果。

2. CloudSim 层

CloudSim 仿真层的主要作用是对基于虚拟化的数据中心环境中的虚拟机、内存、存储、带宽等进行建模仿真。将物理机切分为虚拟机、应用程序管理、集群系统状态监控等工作由 CloudSim 仿真层来完成。用户在 CloudSim 仿真层编写自己的策略，就可以对虚拟化数据中心的虚拟主机分配策略进行研究，评估不同的分配策略下数据中心的运行情况。云应用开发人员可以在 CloudSim 仿真层测试不同的云应用的运行效果。

实际的云计算环境中基本组成元素是数据中心（Datacenter）。数据中心包含了大量的物理主机，且云环境下的物理主机是可以被多个虚拟机共享的，CloudSim 定义了一组资源共享策略的接口（UtilizationModel），来描述如何使用共享资源，CloudSim 中的主机可以被多个虚拟机共享。资源共享策略主要有空间共享（Space-Based）策略和时间共享（Time-Based）策略。

空间共享策略是指在某一段时间内只把计算资源分配给某一个虚拟机/计算任务独占；时间共享策略是指某一时间段内计算资源可以在多个虚拟机/计算任务之间进行共享。例如，一台具有两个 CPU 的主机，CloudSim 在主机上部署了 2 个虚拟机 VM1、VM2，每个虚拟机都有 4 个任务，VM1 上的任务为 t1、t2、t3、t4，VM2 上的任务为 t5、t6、t7、t8，如图 10.2 所示。

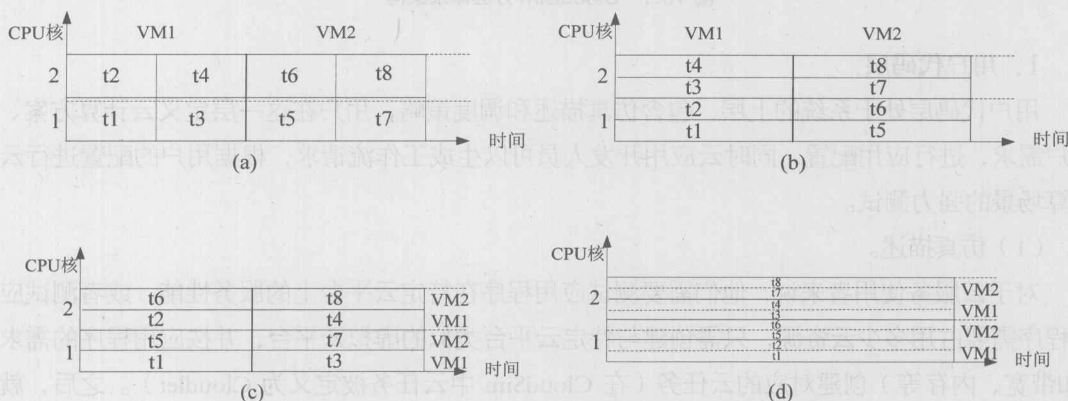


图 10.2　不同资源共享策略下的任务执行情况

图 10.2（a）所示为主机层和虚拟机层都采用空间共享策略的计算任务时间图，VM1 先独占 2 个 CPU，待任务处理完再交给 VM2，同时任务 t1 和任务 t2 分别独占 CPU1 和 CPU2，待处理完成后交给 t3 和 t4；图 10.2（b）所示为主机层采用空间共享的策略，在虚拟机层采用时间共享的策略。图 10.2（c）所示为在主机层采用时间共享的策略，虚拟机层采用空间共享的策略；图 10.2（d）所示为在主机和虚拟机层都采用了时间共享策略。

10.2.3 CloudSim 的使用模型场景

CloudSim 的用途十分广泛，本节讲解 CloudSim 的 3 种典型的使用模型场景。

1．云数据中心的能耗模型

云计算系统包含大量互相连接的主机、存储设备和网络设备等，维持这样庞大的系统运行需要消耗大量的电力。CloudSim 提供了电力控制策略的模拟，能够让用户设计出符合本地数据中心特点的电力方案，从而节约成本，提高整个系统的运行效率。

在 CloudSim 中实现一个抽象类 PowerModel，用来对电力策略进行建模。用户可以通过继承该抽象类，编写自己的电力供应方案，在 CloudSim 上进行仿真实验，从而验证供电方案的整体效果。

2．云平台的经济模型

云计算是基于互联网的服务的增加、使用和交付模式，通常涉及通过互联网来提供动态、易扩展且经常是虚拟化的资源。用户可以像使用水和电一样使用云计算资源，只需付费给云服务提供商就可以租用其提供的计算、存储以及网络等资源。对计算资源、网络资源以及存储资源的定价对于数据中心的运营非常重要。

CloudSim 中对定价策略进行模拟分为基础设施层和服务层两个层次。

（1）基础设施层：这一层主要包括内存单元的价格、外存的价格、数据传输的单位成本以及计算资源的价格。

（2）服务层：这一层主要是应用程序服务使用的资源价格。如果使用者只是利用了云中的基础设施而没有在其上部署任何的应用，比如只是创建了几台虚拟机，并没有在虚拟机上运行任何的任务，那么他将不需要为服务层付费。

CloudSim 的数据中心类（Datacenter）包含了一些关于价格的参数，如 CPU 的使用价格、网络的使用价格、内存和硬盘的使用价格等，方便价格策略的建模。

3．联合云模型

在讲解联合云模型之前，首先需要对以下几个概念进行区分：公有云、私有云、混合云以及联合云。

（1）公有云：面向互联网大众的云服务。其受众是整个互联网环境下的所有人，只要注册缴纳一定的费用任何人都可以使用其提供的云服务。目前，比较流行的公有云平台有国外的 Amazon EC2、GAE（Google App Engine），国内的 SAE(Sina App Engine)、BAE(Baidu App Engine)等。

（2）私有云：面向企业内部的云计算平台。使用其提供的云服务需要一定的权限，一般只提供给企业内部员工使用。其主要目的是合理地组织企业已有的软硬件资源，提供更加可靠、弹性的服务供企业内部使用。

（3）混合云：混合了私有云和公有云。一般像银行这样的单位，其内部的私有云系统在用户访问高峰期的时候很难满足要求，此时就可以接入到公有云中应对更多的用户请求。

（4）联合云：联合多个云服务提供商的云基础设施，向用户提供更加可靠、优惠的云服务，主要针对公有云平台。比如，部署在云平台上的 CDN（内容分发网络）服务，系统存储的数据内容在地理上是分散的，用户也是分布在世界各地。如果 A 国家的用户请求一个分布在 B 国家的数据内容，那么数据就会途经许多路由，增加了网络的时延。联合云能够自动地将用户请求的数据资源迁移到距离用户比较近的云数据中心，提高 CDN 的质量保证。

CloudSim 中定义了云协调器实体（CloudCoordinator）。它不仅负责与其他的云计算平台进行通信，而且负责监控本云平台系统的状态（如任务负载情况、网络延迟情况等）。在整个仿真阶段，云协调器的监控进程始终是活跃的，从监控进程反馈回来的信息为以后进行云平台之间的任务调度作参考。

在进行联合云仿真的时候，两个需要被控制的基本问题是通信与监控。通信问题主要由数据中心通过以事务为基础的消息进程进行控制。监控问题主要由云协调器进行控制。每一个加入联合云的云计算平台都需要实例化一个云协调器实体。云协调器会根据本地数据中心的状态触发 CloudSim 中联合云的任务负载均衡进程。云协调器监控得到的数据是由传感器实体（Sensor）提供的。在每次监控时，CloudCoordinator 都会查询一下 Sensor，判断数据中心的负载是否达到了事先定义好的任务转移条件（如数据中心负载超过多少等）。如果达到了该条件，那么本地的云协调器就会与联合云中的其他云平台的云协调器进行协商，进行任务负载的转移。

10.2.4 CloudSim 使用实例

CloudSim 是基于 Java 语言编写的开源软件，用户使用 CloudSim 和 Eclipse 集成进行云计算仿真实验和开发工作。

1. 下载 CloudSim

登录 http://code.google.com/p/cloudsim/downloads，下载 cloudsim-3.0.3.zip，解压缩。

2. 准备 Eclipse 开发环境

根据用户机器的 CPU 位数，下载相应的 Eclipse 版本并安装。

单击“File” →“New” →“Java Project”，新建 Java 项目，命名为“CloudSim”，如图 10.3 所示。

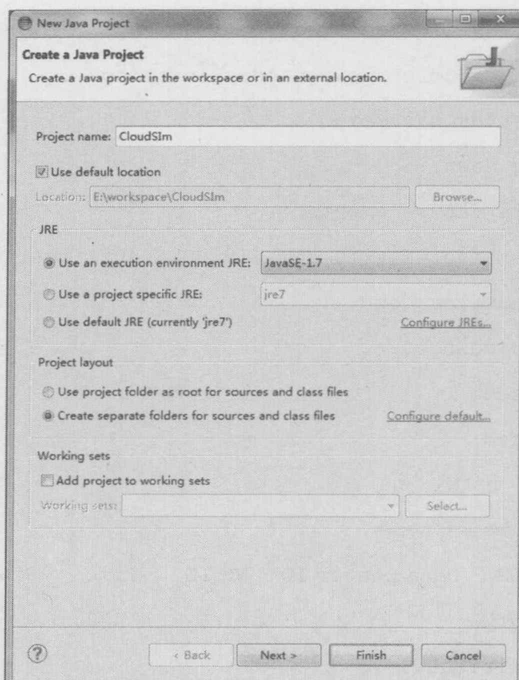


图 10.3　准备 Eclipse 开发环境

工程中用到了 math 里面的类，需要引入 commons-math3-3.2.jar 这个库。用户下载了 commons-math3-3.2.jar 后，选中新建的项目“CloudSim”，单击右键，选择“Build Path” → “Add External Achieve”，将其导入。

3．运行测试程序

Cloud 提供了一些实例程序，使初学者对能快速了解 CloudSim，实例程序存放在解压后的 CloudSim 文件夹中，打开/cloudsim-3.0.3/examples/org/cloudbus/cloudsim/examples，将其中的 6 个示例程序 CloudSimExample1.java~CloudSimExample6.java 复制到工程中。这里我们打开示例程序 CloudSimExample6.java，按“Ctrl+F1”快捷键即可运行示例程序，显示如下的运行结果。

```
Starting CloudSimExample1...
Initialising...
Starting CloudSim version 3.0
Datacenter_0 is starting...
Broker is starting...
Entities started.
0.0: Broker: Cloud Resource List received with 1 resource(s)
0.0: Broker: Trying to Create VM #0 in Datacenter_0
0.1: Broker: VM #0 has been created in Datacenter #2, Host #0
0.1: Broker: Sending cloudlet 0 to VM #0
```

```
400.1: Broker: Cloudlet 0 received
400.1: Broker: All Cloudlets executed. Finishing...
400.1: Broker: Destroying VM #0
Broker is shutting down...
Simulation: No more future events
CloudInformationService: Notify all CloudSim entities for shutting down.
Datacenter_0 is shutting down...
Broker is shutting down...
Simulation completed.
Simulation completed.

========== OUTPUT ==========
Cloudlet ID    STATUS    Data center ID    VM ID    Time    Start Time    Finish Time
    0          SUCCESS        2              0       400      0.1           400.1
CloudSimExample1 finished!
```

4．数据中心仿真实例

本节我们使用 CloudSim 来仿真一个由两台双核物理机组成的最小单元集群，体验 CloudSim 系统的使用。每台物理机分为 4 台虚拟机，即 2 台虚拟机共享 1 个 CPU 核，集群共有 8 台虚拟机，每台虚拟机的运算能力（MIPS）各不相同。这个数据中心需要处理的外部负载任务数为 16。任务调度由 DatacenterBroker 负责，本实例分别使用轮询算法和最小执行时间优先算法进行任务调度，使用 CloudSim 进行数据中心的仿真运行实验，查看两种算法调度的执行情况。这两种算法的区别在于：轮询算法按照虚拟机的顺序，依次将负载分配到虚拟机节点；最小执行时间优先算法将最大的负载分配到处理能力最强的虚拟机。

（1）创建虚拟机。

在 CloudSim 中，我们通过使用镜像大小、虚拟机内存大小、CPU 计算性能、带宽等参数来定义虚拟机的性能。

```
    /** 创建虚拟机的方法 */
    private static List<Vm> createVM(int userId) {
        // 创建一个链表用来存储创建的虚拟机
        LinkedList<Vm> list = new LinkedList<Vm>();
        /* 虚拟机的参数 */
        long size = 10000; // 镜像大小 (MB)
        int ram = 512;     // 虚拟机内存大小 (MB)
        int mips[] = new int[] { 278, 289, 132, 209, 286, 333, 212, 423 };//
虚拟机的CPU性能(MIPS)
```

```
        long bw = 1000;     // 带宽(KBPS)
        int pesNumber = 1;  // 虚拟机的核心数
        String vmm = "Xen"; // 虚拟机的类型
        /*创建虚拟机*/
        Vm[] vm = new Vm[8];
        for (int i = 0; i < 8; i++) {
            // 基于时间共享策略创建虚拟机
            vm[i] = new Vm(i, userId, mips[i], pesNumber, ram, bw, size, vmm,
                    new CloudletSchedulerTimeShared());
            list.add(vm[i]);
        }
        return list;
    }
```

创建外部负载任务，我们对任务的执行长度、占用空间大小、输出文件大小、使用的CPU核数进行定义。

```
    /** 创建云事务的方法 */
    private static List<Cloudlet> createCloudlet(int userId) {
        /*创建一个链表用来存储云事务*/
        LinkedList<Cloudlet> list = new LinkedList<Cloudlet>();
        /* 云事务参数*/
        /* 所需的指令长度(Million 条)*/
        long length[] = new long[] { 19365, 49809, 30218,44157,16754,18336,
20045, 31493, 30727, 31017, 59008, 32000, 46790, 77779, 93467, 67853 };
        long fileSize = 300;   // 文件大小(MB)
        long outputSize = 300; // 输出文件大小(MB)
        int pesNumber = 1;// 使用的核心数
        /* 资源的共享策略*/
        UtilizationModel utilizationModel = new UtilizationModelFull();
        Cloudlet[] cloudlet = new Cloudlet[10];
        for (int i = 0; i < 16; i++) {
            cloudlet[i] = new Cloudlet(i, length[i], pesNumber, fileSize,
                    outputSize, utilizationModel, utilizationModel,
                    utilizationModel);
            // 设置云事务所属的用户
            cloudlet[i].setUserId(userId);
            list.add(cloudlet[i]);
```

```
        }
        return list;
    }
```

主程序是 CloudSim 仿真的重点，用 CloudSim 仿真的主要步骤分为 6 步：初始化 CloudSim 程序包、创建数据中心、创建数据中心代理、创建虚拟机和云事务、开始仿真、打印仿真结果。

```
    /* 创建主函数运行实例*/
    public static void main(String[] args) {
        Log.printLine("Starting CloudSimExample6...");
        try {
            /* 第一步： 在所有实体创建之前初始化 CloudSim 程序包*/
            int num_user = 1;           // 用户数
            Calendar calendar = Calendar.getInstance();
            boolean trace_flag = false; // 是否跟踪事件
            / 初始化 CloudSim 库*/
            CloudSim.init(num_user, calendar, trace_flag);
            /*第二步：创建数据中心*/
            Datacenter datacenter0 = createDatacenter("Datacenter_0");
            Datacenter datacenter1 = createDatacenter("Datacenter_1");
            /*第三步：创建数据中心（用户）代理*/
            DatacenterBroker broker = createBroker();
            int brokerId = broker.getId();
            /*第四步：创建虚拟机和云事务，并将其传递给数据中心代理*/
            vmlist = createVM(brokerId);                // 创建 8 个虚拟机
            cloudletList = createCloudlet(brokerId);  // 创建 16 个云任务
            broker.submitVmList(vmlist);                // 提交虚拟机列表
            broker.submitCloudletList(cloudletList);  // 提交云事务列表
            /* 第五步：开始仿真*/
            CloudSim.startSimulation();
            /* 第六步：仿真结束，并打印仿真结果*/
            List<Cloudlet> newList = broker.getCloudletReceivedList();
            CloudSim.stopSimulation();
            //printCloudletList(newList);
            printCloudletCost(newList);

            Log.printLine("CloudSimExample6 finished!");
        } catch (Exception e) {
```

```
            e.printStackTrace();
            Log.printLine(
            "The simulation has been terminated due to an unexpected error");
        }
    }
```

（2）定义数据中心。

```
    /** 创建数据中心的方法 */
    private static Datacenter createDatacenter(String name) {

        /*创建数据中心的步骤*/
        /* 1.创建一个 Host 链表用来存储数据中心中的主机*/
        List<Host> hostList = new ArrayList<Host>();
        /* 2.主机的参数*/
        int hostId = 0;
        int ram = 2048; // 主机内存(MB)
        long storage = 1000000; // 主机的磁盘大小(MB)
        int bw = 10000; // 数据中心带宽

        for (int i = 0; i < 2; i++) {
        /*3.一台主机可以包含多个 CPU 或核心,
         创建一个 PEs 链表用来存储处理核心或 CPU 的性能参数*/
            List<Pe> peList = new ArrayList<Pe>();
            int mips = 1000;
            peList.add(new Pe(0, new PeProvisionerSimple(mips)));
        /*4.创建主机*/
            hostList.add(new Host(hostId, new RamProvisionerSimple(ram),
                    new BwProvisionerSimple(bw), storage, peList,
                    new VmSchedulerTimeShared(peList))); // 创建 5 台主机
            hostId++;
        }
        /* 5. 创建一个 DatacenterCharacteristics 类用来存储数据中心的属性*/
        String arch = "x86";// 硬件系统架构
        String os = "Linux";// 操作系统
        String vmm = "Xen"; // 虚拟机类型
        double time_zone = 10.0;// 时区
        double cost = 3.0;// CPU 的使用价格
```

```
        double costPerMem = 0.05;// 内存价格
        double costPerStorage = 0.1;// 外存价格
        double costPerBw = 0.1; // 带宽价格
        LinkedList<Storage> storageList = new LinkedList<Storage>();
        DatacenterCharacteristics characteristics =
            new DatacenterCharacteristics(
                arch, os, vmm, hostList, time_zone, cost, costPerMem,
                costPerStorage, costPerBw);

        /*6. 创建数据中心*/
        Datacenter datacenter = null;
        try {
            datacenter = new Datacenter(name, characteristics,
                    new VmAllocationPolicySimple(hostList),
                    storageList, 0);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return datacenter;
    }
```

10.3 云计算系统相空间模型

云计算系统从诞生之日起就与大规模、异构性以及复杂性息息相关。大规模是指云计算系统会涉及多个数据中心、海量的物理节点和网络设备；异构性主要表现在海量的物理节点的软硬件配置各不相同；对于大规模异构的云计算系统来说，无论是系统内部的资源调度和负载均衡，还是对系统的服务性能评估和服务定价都是十分复杂的。使用仿真技术进行云计算系统相关方面的研究可以简化研究工作、节省大量成本。

云计算系统相空间模型是一种针对具有海量节点的云计算集群的仿真模型，通过将物理节点的主要参数（如CPU占用率、内存占用率）抽象出来作为点的坐标，构建云计算集群的相空间。将节点的参数变化转化为相空间中点的运动，利用海量节点在参数相空间的运动与热力学运动的相似性，定义广义温度、广义熵等广义热力学参数，我们可以通过节点在相空间中的投影点来观测节点的状态变化，并通过广义温度、广义熵等的参数来监控集群的整体运行情况。

1．云计算集群的相空间投影

云计算集群中的每台服务器的工作状态可以用一个状态参数向量来表述(参数 1，参数 2，…，参数n)，如（CPU占用率0.3，内存占用率0.2，…，连接数占用率0.1）。当它映射

在相空间模型下时，参数向量的维数对应于相空间的维数，向量终点在相空间的位置代表服务器多个参数的综合负载情况，反映了服务器当前的工作状态。

参数相空间的定义：由服务器的某两个或多个参数为广义坐标轴所形成的二维或多维空间称为云计算系统的参数相空间。参数相空间是云计算集群的整体工作情况在某个时刻上的一个快照。

动量相空间的定义：为了描述云计算的动态工作情况我们定义了动量相空间，动量相空间以投影点在某一单位时间片内在相空间内移动的距离看作速度作为纵轴（投影点的质量设为 1，纵轴就可看作是动量），投影点离当前时刻广义重心的距离作为横轴，一个调度良好的云计算系统在动量相空间中应聚集在原点附近，系统投影点聚集在原点附近表明当前系统运行稳定、负载均衡。

2．云计算集群在相空间上投影的广义参数定义

具有海量节点的高耦合云计算集群向相空间投影后，由于在外部负载请求的作用下相空间上的各个点会不断地产生运动，大量的点的运动呈现出较为典型的热运动特征，分析参数相空间上各点的位置便可以对集群资源进行有效地调度，而且我们可以利用热力学中的参数特性来对应的定义云计算系统在相空间投影下的广义热力学参数作为对云计算系统进行分析的工具。

假定服务器数量为 m，服务器参数有 n 个，由于 n 个参数在调度时的重要性不同，所以假设各参数所占权重分别为 $a_1, a_2, \cdots, a_n$，且 $a_1 + a_2 + \cdots + a_n = 1$，则服务器节点向 n 维相空间的投影点集为

$$A = \left\{ \left(x_{i1}, x_{i2}, \cdots, x_{in}, \right) : 1 \leqslant i \leqslant m, 0 \leqslant x_{i1} \leqslant a_1, 0 \leqslant x_{i2} \leqslant a_2, \cdots, 0 \leqslant x_{in} \leqslant a_n \right\}$$

集群在参数相空间中投影的广义重心位置是判断集群整体负载情况和建立集群管理调度策略的重要参数。云计算集群参数相空间投影的广义重心坐标是由被投影的所有服务器工作状态参数的平均值构成。云计算集群在某时刻投影点的广义重心位置为 G，坐标写为 $\left(X_1, X_2, ..., X_n \right)$. 广义重心坐标计算公式为

$$X_1 = \frac{\sum_{i=1}^{m} x_{i1}}{m}, X_2 = \frac{\sum_{i=1}^{m} x_{i2}}{m}, \cdots, X_n = \frac{\sum_{i=1}^{m} x_{in}}{m}$$

在对云计算系统进行调度效果评价时，相空间上所定义的广义温度和广义熵分别反映了系统当前的活动状态和系统的均衡状态。在实际计算时可以将单位时间间隔内的相空间所有点两次位置采样的平均移动距离作为广义温度。根据热力学原理单个点的运动只有速度概念，大量点的运动就具有了温度的概念，当节点接受新任务时，其相空间投影点向相空间右上方运动，当节点释放任务时，其相空间投影点向相空间原点运动。广义熵的计算一般采用将相空间划分为 $n \times n$ 的网格，根据服务器投影点落入网格的数目 l，利用广义熵的定义公式 $K = \frac{\ln l}{\ln(n \times n)}$ 来计算，分母 $\ln(n \times n)$ 的作用是使广义熵归一化，广义熵的值越大系统越不均衡。我们通常把相

空间上定义的这类热力学参数称为广义热力学参数。

练习题

1. 网格计算仿真系统主要有__________、__________、__________等。
2. GridSim 和 CloudSim 的区别有哪些？
3. 简述 CloudSim 仿真的主要步骤。
4. 编写一段简单的 C 语言程序对由 1000 个同构节点的集群进行模拟定义。

参考文献

[1] 陈国良,吴俊敏,章锋,章隆兵. 并行计算机体系结构[M].北京：高等教育出版社,2002.

[2] 王鹏. 云计算的关键技术与应用实例[M]. 人民邮电出版社, 2010.

[3] Calheiros R N, Ranjan R, Beloglazov A, et al. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms[J]. Software: Practice and Experience, 2011, 41（1）: 23-50.

[4] 孟小峰, 慈祥. 大数据管理: 概念, 技术与挑战[J]. 计算机研究与发展, 2013, 50（1）: 146-169.

[5] The fourth paradigm: data-intensive scientific discovery[J]，2009.

[6] 中国大数据技术与产业大发展白皮书[R].中国计算机学会，2013.

工 业 和 信 息 化 人 才 培 养 规 划 教 材

云计算与大数据技术

Cloud Computing & Big Data

本书全面介绍了云计算与大数据的基础知识、主要技术、基于集群技术的资源整合型云计算技术和基于虚拟化技术的资源切分型云计算技术。全书共10章，主要内容包括云计算与大数据概述、相关技术、虚拟化技术、集群系统基础、MPI、Hadoop、HPCC、Storm、数据中心技术和云计算大数据仿真技术。本书注重实用，实验丰富，将实验内容融合在课程内容中，使理论紧密联系实际。

CLOUD COMPUTING & BIG DATA

TECHNICAL AND VOCATIONAL EDUCATION



人民邮电出版社
教学服务与资源网
www.ptpedu.com.cn

教材服务热线：010-81055256
反馈/投稿/推荐信箱：315@ptpress.com.cn
人民邮电出版社教学服务与资源网：www.ptpedu.com.cn





ISBN 978-7-115-34803-6
定价：32.00 元

封面设计：董志桢